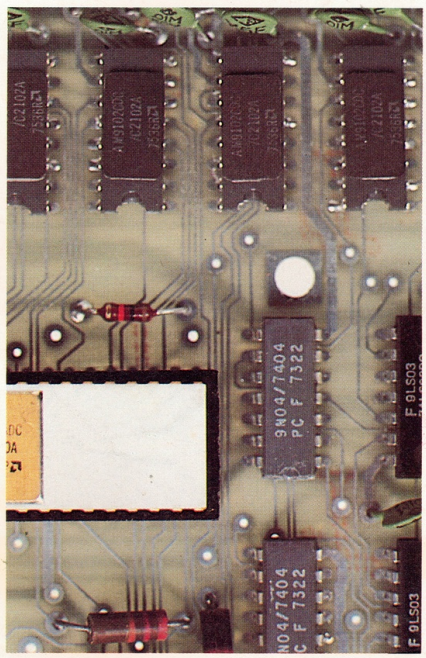
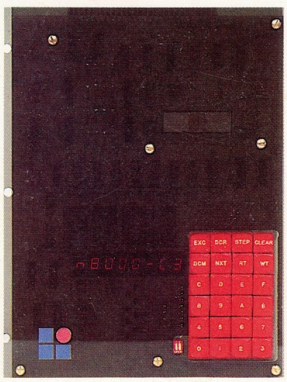
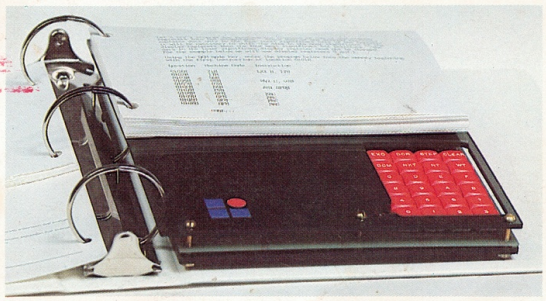
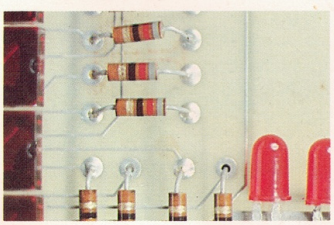
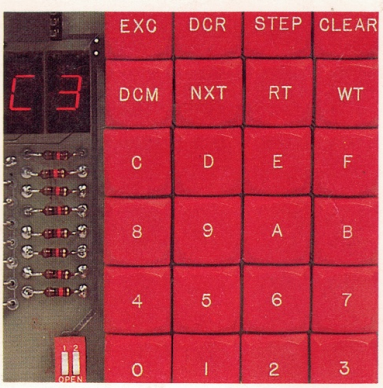
Microcomputer Applications Handbook











iasis inc.

© 1976, Iasis Inc.

MICROCOMPUTER APPLICATIONS HANDBOOK

SECOND PRINTING

David J. Guzeman

President
Iasis Inc.

Iasis Inc.
815 W. Maude Avenue
Sunnyvale, California 94086

Copyright © 1976 by Iasis Inc.

All rights reserved. No part of this book may be reproduced by any means, nor transmitted nor translated into a machine language without the written permission of the publisher.

Printed in the United States of America

Contents

Introduction

1. Microcomputers — Promises and Pitfalls
2. Basic Operation of the Computer
3. The Role of the OEM Microcomputer System
4. The Role of the Development System
5. Hardware Design Casebook
6. Writing the Monitor Program
7. The ia7600 and ia7700 Systems
8. Applications

Appendix A — The Hexadecimal Number System

Appendix B — 8080 Architecture

Appendix C — The 8080 Instruction Set

Appendix D — Data Sheets

Introduction

"The Iasis Series 70 Systems Are Designed to be a Bridge Between the Microprocessor Chip and a Viable Microcomputer System."

Fortune magazine has hailed the birth of the microprocessor as the second computer revolution. Early press releases referred to the microprocessor as the "computer on a chip" and the "fingertip computer" but like much breakthrough publicity, these were overstatements. While the microprocessor did reduce system costs by a factor of ten or more, it still required memory and I/O (Input/Output) circuits to make a complete microcomputer system. The combination of microprocessor and its supporting circuits is a complex interface problem, a fact that has slowed the introduction of the microcomputer into real life applications.

The Iasis Series 70 Systems are designed to be a bridge between the microprocessor chip and a viable microcomputer system. Series 70 Systems make microcomputer technology available at three levels. First the ia7300 Microcomputer Systems offer the engineer an instant-on microcomputer system on a single board. Keyboards, displays and audio cassette interfaces on the boards make the systems self-contained and capable of having programs entered into them directly. Initially offered in an 8080 version the 7300 systems will be expanded in later months to include other microprocessors. The ia7500 workshops provide engineers with hands-on experience in a step-by-step program designed to give them an intimate working knowledge of the Series 70 Microcomputer Systems. The workshops are individual study programs based on the ia7300 evaluation boards and ia7600 OEM boards. Experiments and text materials take the user through each board. Use is made of an oscilloscope to examine individual memory locations, registers, adress and data buses, etc. The workshops are available in either kit form or prewired and tested.

The ia7600 product line is a set of assembled and tested OEM boards and systems available in production quantities. Included in the set of boards are several varieties of single board systems as well as CPU, RAM, PROM, and several I/O boards. Use of opto-isolators and power triacs on some of the I/O boards reduce system interfacing problems to a minimum.

Finally the ia7700 development system provides the engineer with a flexible, easy to use prototyping system for writing and debugging programs. Because this system is made up largely of ia7600 boards, the interfacing power of the OEM boards has been extended to the ia7700, and the transfer from this stage of design to production use of ia7600 systems is virtually instantaneous.

Since the ia 7500, 7600 and 7700 systems all employ exactly the same bus, the user benefits from having all of the systems operate in the same fashion with all of the same bus conventions, etc. Differences are only in quantity of memory and I/O lines, not in differences in function. Even those 7600 OEM boards built around CPUs other than the 8080, can be prototyped and programs written on the 7700. This ensures the maximum amount of versatility within the engineering group since the group is not tied to one type of microprocessor either through familiarity or compatibility with the group's development system.

In a final closing of the circle, the ia7500 workshops can then be used to train field service technicians. As part of this workshop the student builds his own field service test equipment for field installation and debugging of ia7600 systems.

Thus the Iasis System 70 Microcomputers can provide the initial training of an engineering staff, provide them with system development and debugging equipment, supply the actual production OEM systems, and train and equip field service personnel.

On the following pages you will find a detailed description of the entire Iasis System 70 Microcomputer product line. As this is a growing family we hope you will take a moment to fill out the mailing card at the back of the book so we can continue to keep you updated with newer editions of this applications manual.

Chapter One
Microcomputers... Promises and Pitfalls

"The Trend Toward Larger and Larger Logic Blocks Continues . . ."

THE PROMISE OF THE MICRO-COMPUTER. Traditionally engineers have attacked the design of digital logic systems in a relatively random fashion that relied on intuition, inspired hunches and many paper redesigns. Each system was made up of logic elements – gates, flip-flops, etc., combined into a workable system. Typically the designer worked on the entire system at once, doing a little here, a little there, filling in holes in the logic design in a scattered, piecemeal fashion. As each problem was solved, it affected the rest of the system which then had to be modified to accept the new subsystem. This process continued until the design was complete and the system worked.

Each system was different in the same way that every city is different. While each city is made up of common elements – houses, apartment buildings, office buildings, etc., the way these elements are interconnected differs drastically from one city to the next. And so it is with digital logic systems. Each has a unique layout, and while each may be made of the same building blocks, the interconnections are completely different, so that each system has its own personality, its own data flow, and its own interfacing requirements. These digital systems are said to be implemented with random logic.

This type of system places a large burden upon the shoulders of the design engineer since each system must be designed from the ground up. The problem was compounded by glitches resulting from some paths being longer and hence requiring more time than others. Signals that were supposed to arrive at the same place at the same time sometimes didn't, and when this happened the resulting system failure was attributed to "race conditions." Naturally good designs eliminated potential race conditions, but it was not always easy to foresee them. Sometimes the system prototype worked well, but the production model failed because of unpredicted race conditions. Needless to say, the design and development of a typical digital system required many months of expensive engineering time. In addition to the direct engineering expenses were hidden costs caused by delays in getting the product to the marketplace.

The actual implementation of the digital system relied heavily on the state of the semiconductor technology. Digital systems did not become practical until the advent of solid state electronics due to the large number of active devices required for a typical system and the short life and inherent unreliability of the vacuum tube. The systems of the 1950's made use of discrete components – individual transistors, diodes, resistors, etc. By the 1960's the introduction of the integrated circuit had revolutionized digital electronics.

By packing a whole set of logic gates or a pair of flip-flops into one plastic package with 14 leads the number of connections to the printed circuit board could be reduced by a factor of ten or more. Early studies proved that the already low failure rate of solid state electronics was directly related to the number of system interconnects, so the new integrated circuits represented a significant jump in system reliability.

The ten-fold reduction in the number of components that had to be tested and assembled resulted in a lower system cost. Further savings were realized because the cost of the integrated circuit was much lower than the cost of the discrete components that it replaced. It was a situation in which everyone won. Further advances in the semiconductor art allowed larger and larger blocks of logic to be placed in each integrated circuit. These were referred to as MSI, Medium Scale Integration. By the early 1970's the use of MSI was universal and was resulting in new types of products that were earlier thought to be impractical.

The trend toward larger and larger logic blocks continues, the present state of the art being epitomized by 4096-bit Random Access Memory circuits, RAMs. Components of this complexity are referred to as LSI (Large Scale Integration). There are actually three types of LSI memories being widely used today. While RAMs were memory circuits that could store data in a scratchpad fashion, that is, they could accept and hold data and then upon command read it back to the system, ROMs, Read Only Memories, have their data built into them permanently at the time of manufacture. Initially these devices were used to hold tables and blocks of data that did not change. Later they were also used to simulate blocks of

"Dedicated Microcomputer Systems Are Now Being Used in Products Impractical Last Year . . ."

logic that did not lead themselves to replacement by standard MSI or LSI blocks of logic. The large tooling costs involved with ROMs made them practical only in large volume applications. This ultimately led to the development of the PROM, Programmable Read Only Memory, a device that could be programmed by the user in a special piece of programming equipment. This made prototype and small quantity usage of the ROM principle practical in new systems. A variation of the PROM is the EROM, Erasable Read Only Memory, which is programmed like a PROM but can be erased by exposure to ultraviolet light and loaded with a new or corrected program.

It is important to realize at this point that all of the systems discussed have been implemented with random logic. Every system was different, and while RAMs tended to be used in arrays, ROMs and PROMs were usually sprinkled around the system. The worries about race conditions, clock skew and a host of other digital gremlins continued to plague the system engineers.

The introduction of the microcomputer represented a total change in the design approach to digital systems. Through the extensive use of LSI semiconductor components, large blocks of the system came prefabricated and could be connected in a predetermined standard fashion. The system that resulted was analogous to building a model city by hanging standard subdivisions of varying sizes onto a main arterial. Depending on the nature of the application a system might have a larger memory or perhaps more I/O channels, but in each case the functional blocks are connected to the main arterial in a standard way. Systems differ from one another only in the size of the blocks.

This greatly simplifies the problem of the designer. Gone are the giant logic diagrams, race conditions, clock skew and all the other problems that plagued the designer of random logic systems. In its place are a set of functional blocks, CPU, memory and I/O, that differ from system to system only in size. The hardware aspects of the designer's task are reduced to near zero. There are other dimensions to the problem, however.

A random logic system can be thought of as a set of inputs and outputs with conditioning logic in between. The conditioning logic accepts data on the system inputs and uses it to generate a set of unique output signals which are sent to the system outputs. A microcomputer works in the same way except the conditioning logic is replaced by a program stored in the system memory. The CPU, Central Processing Unit, then refers to that program to generate output signals according to the data arriving on its inputs. The thing that distinguishes one system from another is the program stored in memory. The task of writing that program falls on the design engineer.

Because the cost of memory bits is substantially less than the MSI they replace, the resulting microcomputer will cost less than its random logic equivalent. The old argument about MSI and LSI costing less than individual transistors or gates reappears in a more modern form. The microcomputer will cost less, contain fewer parts, fit on a smaller board, consume less power and be more reliable than its logic equivalent. And once the designer is familiar with writing programs he can design a microcomputer system much faster than a random logic system.

Because the personality of the microcomputer is contained in its program (which is in turn stored in integrated circuit ROMs or PROMs) design changes are easy to implement by merely changing ROMs. Products can be updated or features added quickly and cheaply.

The promise of the microcomputer is in its low cost and ease of design. Dedicated microcomputer systems are now being used in products impractical last year, from digital watches to electronic games.

THE PITFALLS OF THE MICROCOMPUTER. Unfortunately, the promise of the microcomputer has only been realized by a few. This is no fault of the product itself but of the support systems built up to service it. To appreciate the nature of the problem, it is necessary to understand the process a design engineer goes through to design and develop a microcomputer system.

Most engineers first learn about microcomputers through one of the many trade magazines or electronics shows. Each month advertisements and magazine articles extol the virtues of the microprocessor and allude, sometimes not too subtly, that any reader expecting to stay current in the field of solid-state electronics should be working with these exciting new devices. Unfortunately there is very little literature sufficiently detailed to act as a design guide and at the same time organized in a way to painlessly introduce the subject at the engineering level.

Usually at this point the frustrated engineer decides to purchase some sort of working system. He intuitively feels that armed with his trusty oscilloscope he can figure out an operating microcom-

"Iasis Series 70 Systems Can Make the Promise of the Microcomputer Come True . . ."

puter system. The description of the various systems on the market are anything but clear, adding greatly to the confusion. These systems usually consist of a set of compatible printed circuit boards, although the interconnections between the boards are left to the user. There are other problems. The TTL compatible I/O ports could interface to other digital systems but not to any of the loads that the system would actually be driving such as solenoids, relays, hammer drivers, LED displays, etc. The design of the inteface circuits is left to the user.

Then there is the problem of getting information into the system. Teletypes are commonly used for this purpose but they generally require special interface circuits that are seldom supplied with the boards. Even CRT terminals and TTL compatible keyboards require special "interpreter" programs called monitors written so the system can understand the data arriving at its input ports. These programs are not supplied with the boards making it necessary for the user to develop his own.

Writing programs is the biggest problem of all. The board sets contain no provision for programming PROMs or EROMs. It is left to the user to supply preprogrammed PROMs; this requires the purchase of a PROM programmer. Finally there is no way to write and debug programs using the powerful techniques possible in assembly language programming. Instead the user has to work in machine language, a practice that is both tedious and prone to errors.

The larger development systems are more expensive, upwards of $10,000, but they offer the powerful tools of assembly language programming. These systems are usually built around a large block of RAM memory and some PROMs that contain special monitor programs used to assist the operator in writing and debugging his own programs. Interface circuits to a teletype are provided so that programs can be written on the teletype, and then entered into the system RAM where they can later be executed. Since the program is contained in RAM it can easily be modified or corrected until the user is satisfied with its execution.

In this mode the development system operates as a simulator. It offers a set of TTL compatible I/O ports that can be interfaced to outside devices. Of course, the system can not be turned off without losing the new program that has been written into RAM storage. But this is not a serious disadvantage since the development system was designed only for the writing and simulating of programs and is capable of storing programs on punched paper tape while other programs are being debugged.

The RAM section of the memory also allows a special intermediate program called an assembler to be written into the system so that programs can be written using assembly language. When the user is satisfied that the program is operating in the system correctly, he can load it into PROMs. These PROMs can then be plugged into sockets on the memory board of the small board set already discussed. If sufficient care has been exercised the board set will then perform the program in the same way that the development system did.

In principle, the combination of development system and OEM cards is a perfect marriage. In fact, there are several things wrong with the systems presently available.

At this time the documentation provided with these systems is woefully inadequate and really leaves the user on his own to get his system up and running. Once this happens, however, he is faced with a host of other problems. To communicate with his system he must purchase a teletype and then modify it so it can talk to the system. In order to program PROMs a separate PROM programmer is required. The use of the system to drive peripherals is difficult to say the least. None of the I/O ports are set up to handle anything but other TTL logic; there are no relay, opto-isolator, or triac boards offered so the user must provide his own interface circuitry if he chooses to use the development system in any kind of real world situation.

These problems, plus the high initial cost of the development system, have discouraged many potential microcomputer users resulting in an image of microcomputers as only being worthwhile in large volume applications. This is really unfortunate since the microcomputer IS all it's made up to be; it's the support systems and literature that are deficient.

These were the problems that existed when the design of the Iasis Series 70 Microcomputer Systems were undertaken. This system can make the promise of the microcomputer come true, even in applications requiring only a few systems.

Chapter Two
Basic Operation of the Computer

The invention of the modern electronic computer has had a dramatic impact on the day to day lives of everyone of us, the extent of which will probably not be recognized for some time. The intent of this chapter is to give the reader a short, working description of the parts of a computer and how it operates. Since the differences between large, general-purpose computers and mini or even microcomputers are basically quantitative rather than qualitative, we will not describe any particular computer or class of computers but will keep our introductory remarks general. In the second half of this chapter we will then describe the ways in which microcomputers differ from computers in general.

For the purpose of this discussion, we can think of every computer as made up of four basic parts. These are the arithmetic unit, the memory, the input/output circuits (I/O) and the control unit. The control unit generates control and timing signals that are sent to the other sections of the computer which causes them to move and manipulate data according to the wishes of the user. The user must, of course, specify to the computer exactly what his wishes are. He does this with binary numbers that represent instructions.

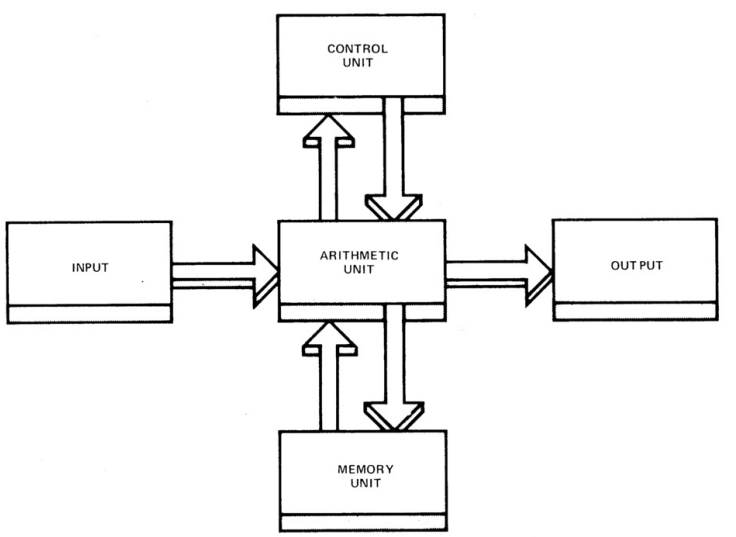


FIG. 2-1. Every computer consists of four basic parts: the arithmetic unit, the memory, the input/output circuits and the control unit.

Each instruction is made up of a binary word or number that is recognized by the control unit and causes it to respond by sending out the appropriate timing and control signals, one set for each instruction received. When the user combines a list of instructions so that the computer can perform more complex tasks, the sequence of instructions is said to make up a program. As this sequence is presented to the control unit, one instruction at a time, the control unit will generate a corresponding sequence of control signals.

Originally the program is generated by the human programmer in the form of written instructions. These are translated into binary number codes and stored in the computer memory. The memory can be visualized as a set of storage boxes, numbered sequentially beginning with box #0, each containing one instruction. The program is then stored in the boxes or memory locations, with the first instruction in box #0, the second instruction in box #1, the third in box #2, etc.

Inside the control unit of the computer is a binary counter called the program counter. When the system is first energized, or after it has been reset, the program counter is reset to the binary equivalent of zero. As a first step in the computer cycle, the contents of the program counter are sent to the memory as an address. The memory searches through itself until it finds the specified memory location, pulls the contents of the box out and sends it back to the control unit. During the first computer cycle, the program counter will contain zero so that box #0 will be searched. In that box is the first instruction of the program. The control unit receives the instruction, decodes it and sends out the appropriate control and timing signals. These signals cause the computer to respond in some predetermined way to the instruction; the computer is said to have executed the instruction.

After the first instruction has been executed, the program counter is incremented and the next address sent to the memory. After being incremented the program counter contains the binary equivalent of one, so it is the contents of memory box #1 that are retrieved from the memory and returned to the control unit. This box contains the second instruction of the program which is then

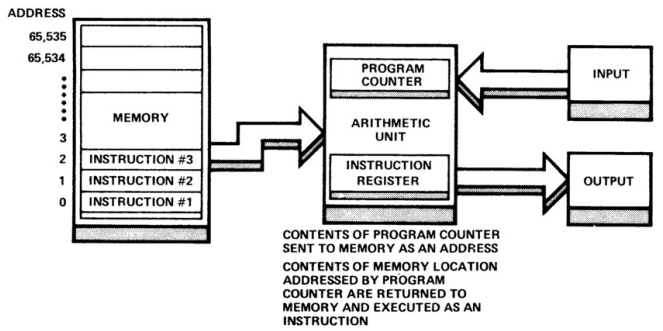


FIG. 2-2. The instruction fetch cycle retrieves the program from memory, instruction by instruction.

executed by the computer. The system proceeds in this fashion, sending out the contents of the program counter as an address, receiving back the instruction at that address and then executing the instruction. Since the program counter is incremented after the completion of each computer cycle, the instructions are retrieved from the memory in the same order they appeared in the program, and hence, the execution of the program occurs in the correct order.

The process of sending out the contents of the program counter and retrieving the instruction from memory is referred to as the instruction fetch cycle. It together with the instruction execution cycle make up the complete computer cycle. It should be noted that the operation of the instruction fetch cycle is an automatic process within the computer and is totally independent of which instruction is being fetched. The instruction execution cycle, on the other hand, depends entirely on the instruction just fetched. For instance, an ADD instruction will be executed in a totally different way from an INPUT instruction.

The actual execution of the instruction takes place primarily within the arithmetic unit of the computer. In most microcomputers all data must flow through the arithmetic section as it is moved about the microcomputer. Thus to move data from one location in memory to another, it is necessary to move the data from the source location into the arithmetic section with one instruction and then into the destination register with a second instruction. Thus the arithmetic section is involved in the execution of most of the instructions. The arithmetic unit generally contains several temporary storage registers organized to serve as a scratchpad memory. These are usually easier to read and write than the regular memory locations, and serve to simplify data flow within the computer. One of the registers called the accumulator has provisions for adding other numbers to its contents, shifting the contents left or right, complementing the contents and indicating if any of these operations result in an overflow.

A computer would be of little use if there was no way to communicate with it. Data is sent to and received from the computer via the Input/Output circuits or I/O "ports." Data presented to the computer at one of its input ports will be read into the system by executing an INPUT instruction. Likewise the computer can cause data to be sent out through one of its output ports by executing an OUTPUT instruction.

All of the operations that occur in the system, memory read or write, arithmetic data manipulation and Input/Output operations, are controlled directly by the instructions the system executes. Thus a program or list of instructions is a detailed activity list for the computer that causes it to perform the desired operations.

The memory is used for two different purposes in the computer. As we have already seen, it is used to store instructions that make up the program, and is accessed sequentially during the instruction fetch cycle starting with location zero. A second use of the memory is to store data used by the computer when the amount of data exceeds the capacity of the scratchpad registers in the arithmetic section.

Notice that as far as the memory is concerned there is no difference between data and instructions; both are simply binary numbers stored one to a location. How does the system then know whether any particular memory location contains an instruction or a data word? It doesn't; only the person programming the computer can keep them separate.

Since the program counter is automatically reset to zero when the system is first energized, it is natural that the first program instruction be placed in memory location zero. In that way the program will begin to be executed when the system is first energized. Since the program is to be executed sequentially

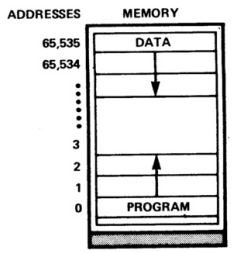


FIG. 2-3. The computer stores the program in the low order addresses of memory and data in the high order addresses.

it will begin at the low order addresses of the memory and proceed up the address scale until the end of the program is reached. All of these locations will be used in instruction fetch cycles and will always contain instructions, never data. Above this point, however, the memory is free and unused and can be used to store data. In writing programs the programmer will always know how much memory is needed for a given program. All he has to do is count the instructions in his program. A program with 1000 instructions requires 1000 memory locations. Any locations above and beyond the 1000 necessary to hold the program can be specified by the programmer as data storage locations.

Computers used for industrial or control purposes are designed to interface with a wide variety of peripheral equipment, some of which may be relatively slow in supplying data to the computer. If the computer is idle while waiting for data from one of these devices, the system will be inefficient. In effect, we are dragging the computer down to the speed of the slowest peripheral in the system, despite the fact that there are many other operations the computer could be doing in the meantime. An example of this would be an A/D converter supplying a new data word to the computer every 100 ms or so. During the time between data words the computer, which can execute a typical instruction in less than 5 μs, could execute 20,000 instructions. Yet, if it is tied to reading and waiting for the data from the A/D, this time will be totally lost to the user.

Practical computers are designed so they can be interrupted by a peripheral device whenever the peripheral has a new data word ready to send to the computer. The interrupt causes the computer to stop what it is doing, read the data word in through an interrupting input port and then return to its original task. In this way the peripheral gear has little or no effect on the efficiency of the system. Of course, when the computer is interrupted it has to know what to do with the data from the interrupting peripheral. This information is contained in an interrupt service program written especially for this task. When an interrupt occurs, the program counter is loaded with the address of the first instruction in the interrupt service program. Thus the next address that is sent out in the instruction fetch cycle is the address of the beginning of the service program. Effectively the computer jumps to that program.

The service program can be quite involved causing all kinds of operations to be performed on the data. Of course, unless we take special steps to keep track of where we are, the computer will never be able to return to the original program. Thus the first part of the interrupt service routine is aimed at storing in memory the contents of the scratchpad registers, including the program counter. When the data has been processed, the contents of the registers are stored and continuity of the main program is assured.

Interrupts are impcrtant for servicing peripheral equipment that is considerably slower than the computer. Sometimes the problem is the reverse, that is, the peripheral is much faster than the computer. In most systems, the throughput is limited by the instruction fetch cycles which must be done several times to read data into the memory from a peripheral. The memory itself is considerably faster. Thus most computers have provisions for DMA, Direct Memory Access, a technique for reading and writing data into the memory without going through the complex instruction cycles. DMA effectively bypasses the arithmetic unit and allows the peripheral to communicate with the memory at a rate many times faster than the normal INPUT/OUTPUT instruction sequence allows.

Most of the computers available today have in common all of the features we have been discussing. In fact the similarities between the various classes of computers far outnumber the differences, which tend to be quantitative rather than qualitative. Thus one of the biggest differences between large, general-purpose computers and mini or microcomputers is the length of the binary word that is recognized and processed by the computer as an instruction. Large computers use 32 or more bits to make a word; minis use 8–16 bits and present microcomputers use 4–8 bit words. Longer instruction words allow larger, more complex instruction sets. As a result programming on the larger machines tends to be easier. Likewise the larger the computer the more memory it uses; maxi computers use megabyte memories. Microcomputers, on the other hand, may use 4096 bytes or less.

Both maxi and minicomputers have general-purpose memory and I/O circuits. As a result any given minicomputer could be used for many different applications merely by writing a new program into the memory. The systems designer using minicomputers needed only to design interface circuitry and write programs.

Were it not for the introduction of the microcomputer this state of affairs would still exist. Microcomputers have one dis-

tinguishing feature that sets them apart from mini and maxi computers and has caused users to change the way in which they use and think about computers. That essential difference is that microcomputers are designed to have permanent, dedicated programs. Each microcomputer is designed to answer a specific need, solve one single problem, handle one and only one application. Because the microcomputer must execute the single program in its repetoire every time the system is energized, that program is stored in special Read Only Memories, ROMs, that retain the program even when their power source is removed. Because each microcomputer is designed for a specific application, unused memory is wasted memory. Microcomputers have exactly the right amount of memory and specially designed I/O circuits for the task at hand.

Unfortunately programs never seem to run correctly the first time. When we were working with minicomputers containing RAM memory this was no problem; we just kept writing new programs into memory replacing the old program until the system worked. But microcomputer programs are stored in ROMs which are loaded by the semiconductor manufacturer; changes are expensive and take time. This has led to the development of special systems used for writing and debugging microcomputer programs without resorting to ROMs. Only when the program runs correctly will it be commited to ROM.

Chapter Three
The Role of the OEM Microcomputer System

". . . prebuilt microcomputer the most cost effective solution in applications requiring 10—100 systems."

The relatively low cost of the microcomputer has allowed it to be used in many applications that were formerly uneconomical when implemented with either minicomputers or random logic. Much of the low cost of the microcomputer system must be attributed to the fact that the microcomputer uses a dedicated program that cannot easily be changed by the user. This restriction has caused an evolution of specialized hardware and software development tools to aid the user in writing trouble-free programs; these systems are the subject of Chapter Four. It has also brought about a different set of hardware/software cost tradeoffs for the user.

Every system has several cost components that determine the total cost of the system. The three major cost components are the engineering, software or programming costs, and the hardware costs. Both engineering and programming represent one-time costs incurred at the development time of the product which are then amortized into the total number of systems used. The hardware costs consists of the cost of the ICs and printed circuit boards as well as the labor costs to assemble and test them.

In general the user may attack the problem of getting a complex digital system into operation in three different ways. The first is to purchase an operating minicomputer system. Even though the user must do his own programming the extensive software support systems available with the minicomputer reduce the programming time to a minimum. On the other hand the general purpose architecture of the minicomputer and the large, expensive RAM memory make this the most expensive approach from the hardware standpoint. The engineering costs are, of course, zero since the system is purchased in an assembled configuration. The overall cost effectiveness of the minicomputer system is determined by the number of systems needed. For applications requiring less than ten systems it is undoubtedly the least expensive and fastest solution to the problem.

The second approach to the problem is to purchase assembled microcomputer systems. Originally offered in sets of compatible printed circuit boards and more recently in the form of single board computers, these systems offer the user the convenience of preengineered, operating systems needing only a program to be placed into the user's application. Because of the requirement that the program be contained in ROM or EROM components, these systems are generally sold with some sort of compatible development system used for writing and debugging the program that will ultimately be executed by the OEM system. While the software support systems used in conjunction with the development systems are not as extensive as those of the minicomputer, they nevertheless offer the user a tremendous advantage over the unfortunate who must program his systems directly in machine code. As before, the engineering costs of the prebuilt microcomputer system are zero. Now, however, the user enjoys a much lower hardware cost at the expense of slightly more software costs. The overall impact of the savings is to make the prebuilt microcomputer the most cost effective solution in applications requiring 10-100 systems (some analyses show the prebuilt microcomputer to be cost effective up to 500 systems).

Any prebuilt microcomputer represents a compromise between the user's application and a general purpose microcomputer designed to handle the greatest number of applications. This is particularly true in the areas of memory size and the number and combination of I/O ports. Invariably the user will end up with extra ports and unused memory. The misfit of memory to the application can be minimized by using sockets for the memory ICs, so that unused RAMs are simply removed from the board. In a similar fashion current designs for I/O ports allow the user to select whether each port is to be an Input or an Output. This results in the optimum I/O structure for most applications.

In any event no matter how well the prebuilt system is optimized there will always be a certain amount of logic on the board that is unused in certain applications. That logic represents a cost to the user that he could save by designing his own system for a particular application. Offsetting the cost savings is the cost of

engineering. The user must amortize the cost of engineering over a smaller number of systems than the manufacturer of prebuilt systems who is selling to a large number of users. As has already been pointed out, the economic cross over point at which the user should design and build his own systems is generally between 100 and 500 systems.

THE SINGLE CARD COMPUTER

There are two possible approaches to the problem of the prebuilt OEM microcomputer system. The first is to construct a single printed circuit board containing microprocessor, system clock, memory and a set of I/O ports. This approach is probably best for small systems using only 4K to 8K bytes of memory. The limited amount of memory places less demands on the addressing capability of the computer and shortcuts in the address decoding can be made. Likewise the address and data bus drivers can be smaller and cheaper.

The advantages of the single card system can best be seen when compared to multicard systems. Use of a single card eliminates expensive connectors and hardware needed to tie multiboard systems together. Multiboard systems require card select logic on the address lines which is eliminated when only one card is used. Bus signals that must be available to all of the cards require extensive buffering; they are usually defined to be active low which sometimes requires extra logic to invert incorrect logic levels. None of these considerations apply to the single board system.

Before leaving the single board computer it should be pointed out that there are no true single board systems on the market in prebuilt form. The reason is simply that no manufacturer can anticipate the I/O requirements of the user; the best he can do is supply him with TTL compatible I/O ports. It is left to the user to interface the TTL levels to reed relays, triacs, hammer drivers, CRT displays, etc. Since there is no room to do this on the single board, a second board designed by the user is required. Unfortunately manufacturers of single board systems have given little thought to the hardware aspects of the second board and how it will be mounted or interconnected to the microcomputer.

THE MULTIBOARD SYSTEM

Systems requiring more than 8K of memory or specialized I/O requirements are more conveniently built up out of several boards. The typical set of boards consists of a CPU board containing the microprocessor, system clock and address and data bus buffers, a RAM memory board, an EROM or PROM board, and a set of I/O boards. When a combination of these boards is plugged into a bus a working system results. The advantage to this approach is that the system can be easily expanded by adding more memory and I/O boards. Furthermore the system is easily customized to specific applications by combining selected boards. In addition, the card sets usually include a universal prototyping board which can be used to hardwire up special I/O boards for specific applications.

Both systems have their advantages; the choice of which to use is generally dependent upon the complexity of the application and the amount of memory required. Both types of systems are included in the Iasis ia7600 product line. Both are compatible with the ia7700 development system, a fact that allows the user to postpone the decision as to which system to use until after the program has been written and debugged.

HOW MUCH MEMORY?

The first question confronting the designer of a microcomputer designer is the amount of memory to be used in the system. In the case of the system that is being designed for a single application this question is automatically answered as soon as the program is completed. In the case of the single board system that must serve many different applications, the question is of prime importance because the memory components occupy a large portion of the board area, and hence determine the size of the board. In themselves they represent one of the largest cost components. The question is further complicated since the OEM system will use a combination of RAM and EROM and this ratio must also be established at the same time the total memory size is determined.

Here we must define whether the system is to be used in some sort of hobby mode as a pseudo minicomputer or whether it is to be a true OEM microcomputer to be used in a dedicated application. Hobby systems require that the user be able to write programs into RAM storage, execute them and write new programs. This requires that the bulk of the memory be RAM with only enough ROM or EROM to hold the system monitor program. Monitor programs can be written in as little as 512 bytes of EROM; complicated versions may require as much as 2048 bytes. The amount of RAM is an open-ended question and is totally dependent upon the complexity of the programs the user is expect-

ing to execute. As a guide, it should be pointed out that it is possible to write a program for a complete floating point arithmetic package in about 750 bytes of memory.

Assuming that the system under question is to be used in an OEM application and not in the hobby mode, the answer to the question of how much memory is totally different. Now it is assumed that the program is contained totally in ROM or EROM and the system RAMs are used only for scratchpad memory locations during the execution of the program. While it is always possible to find exceptions to the rule it generally requires at least 4 times as much EROM as RAM. A typical configuration results in a system containing 1024 bytes of RAM and 4096 bytes of EROM. With the present state of the semiconductor art, this would be implemented easily in 8 RAM components and 4 EROM packages. In a typical system of this size the memory would occupy about 35% of the board. Reducing the memory would not greatly reduce the size of the system but would make the board unsuitable for many applications.

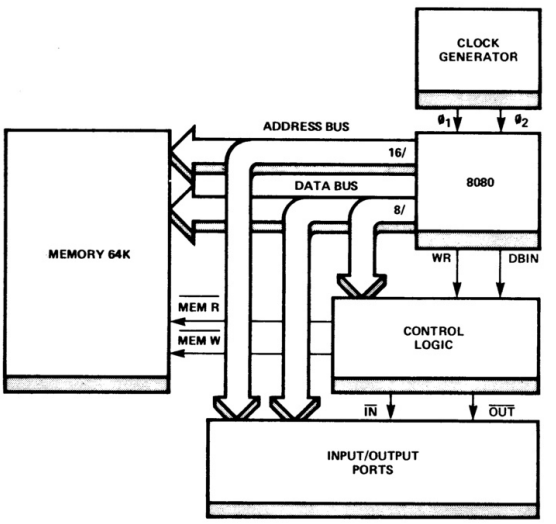


FIG. 3-1 Dedicated I/O configuration uses control logic to generate IN and OUT control signals to enable the Input and Output ports whenever Input or Output instructions are executed.

DEDICATED I/O

In general there are two ways of handling the problem of enabling the I/O ports. The 8080 uses seperate Input and Output instructions to send and receive data from special ports which are enabled by a combination of a control signal and a port address. The control signal is derived from the status word sent out by the CPU at the beginning of the cycle. Thus whenever an Input or an Output instruction is to be executed the appropriate $\overline{IN}$ or $\overline{OUT}$ bus line is pulled low. When this happens the I/O decoding circuits are enabled which allow the 8 low order address lines to select one of the I/O ports. With 8 lines used to address the ports as well as separate $\overline{IN}$ and $\overline{OUT}$ control signals, it is possible to select one of 256 Input ports and one of 256 Output ports, 512 in all.

The 16 line address bus used by the 8080 allows the user to directly address up to 65,536 bytes of memory and 512 I/O ports. However, we have already seen that typical applications require only a few thousand bytes of memory. It is, in fact, difficult to imagine an application requiring more than 16K bytes of memory, let alone the 65,536 byte capability provided us by the 8080. The reason the 8080 has this capability is one of convenience to the IC designer and not out of necessity for the user. The 8080 is an 8-bit CPU; all of its registers are 8-bits long. If one of the registers is used to store and send out an address, the system would be capable of addressing only 256 bytes of memory, not enough for practical applications. If two registers are used to specify an address, the system is capable of addressing 65,536 locations, and it is this system that is used in the 8080. As we will see shortly it is possible to give up some of this huge addressing capability to gain programming flexibility.

MEMORY MAPPED I/O

The addressing and I/O port selection scheme used by the 8080 results in the largest accessible memory. In addition to the extra logic required for this configuration, the dedicated I/O has one very large disadvantage; it requires separate Input and Output instructions to communicate with the I/O ports. In the case of the 8080 an Output instruction is executed by transferring the contents of the accumulator to the Output port specified in the second byte of the Output instruction. Similarly, Input instructions are executed by transferring the contents of the specified Input port to the accumulator. In other words, all communication with the I/O ports

must be made through the accumulator.

Other processors, most notably the 6800, use a different configuration for enabling and selecting the I/O port. This is a technique known as Memory Mapped I/O in which I/O ports are treated as memory locations. This allows any memory write or memory read instruction to act as an I/O instruction which can result in shorter programs that are easier to write and faster to execute.

This same technique can be used with the 8080. The result is greater programming freedom and versatility at the cost of a few memory locations. It can potentially offer more I/O ports than the dedicated I/O configuration, although this feature is seldom significant. A relatively simple way to implement memory mapped I/O with the 8080 is seen in Fig. 3-2. Here only 15 of the 16 address lines are used for accessing the memory; this allows the system to address up to 32K bytes of memory. The final address line is used to enable the I/O ports. Theoretically this would provide the system with 32K I/O ports, far more I/O capability than even the most extreme system would ever use.

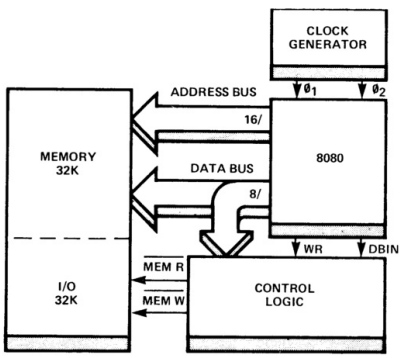


FIG. 3-2 Memory Mapped I/O treats I/O ports as memory locations. This simplifies the control logic.

An example of the I/O decoding circuits for a simpler system using only 16 I/O ports is illustrated in Fig. 3-3. Here again, the high order address line is used to enable the I/O ports, but now only the four low order address lines are used by the decoding logic of the port selectors. There are many possible decoding circuits that can be used for the I/O ports. If desired, all of the lines can be decoded so that, for instance, a system could be configured that addressed 64,536 bytes of memory and 1000 I/O ports. If even this small amount of memory seems too large a sacrifice, it should be pointed out that at the present price of floppy disk memories, it is highly unlikely that a system will ever be built using the full addressing capability of the 8080.

The real reason for using memory mapped I/O techniques is not for the slight reduction in the logic of the control unit; it is to take advantage of the full set of memory read and write instructions to ease the programming burden on the user. Now instead of only IN and OUT the user has 17 instructions at his disposal to read or write into the I/O ports. See Fig. 3-4. These instructions allow the user to perform operations on the data during the very process of writing or reading the I/O ports. We will now consider some I/O port circuits that might arise in actual practice.

INTERRUPTING INPUT PORTS

The system designer frequently finds himself interfacing to a piece of peripheral equipment that is very slow compared to the relatively fast pace set by the 8080. This is the case, for instance, when reading data from a keyboard. The 8080 can process literally hundreds or even thousands of instructions while waiting for the next key depression. In a very simple, dedicated application there is no real reason not to let the system idle reading and rereading the keyboard continually checking for key closures. In more complex systems there are other jobs or programs the CPU could be executing while waiting for a key closure. One example is a system interfacing to a number of keyboards and processing the data from all of them. In this case some way must be found to both read the keyboard and execute another program at the same time.

The normal way of handling this problem is to make use of the interrupt capability of the CPU. The keyboard is set up so that every key closure causes a strobe pulse which is applied to the interrupt input of the 8080. This causes the 8080 to enter an interrupt service program which saves the contents of the working registers in specially allocated memory locations, and then to read the keyboard input port and process that data. At the conclusion of this subroutine the registers are restored to their former contents and control returned to the original program at exactly the point it was interrupted.

If there is to be only one signal that interrupts the system, the hardware is very simple. All that is required is a single flip-flop to reclock or synchronize the incoming interrupt request with the timing

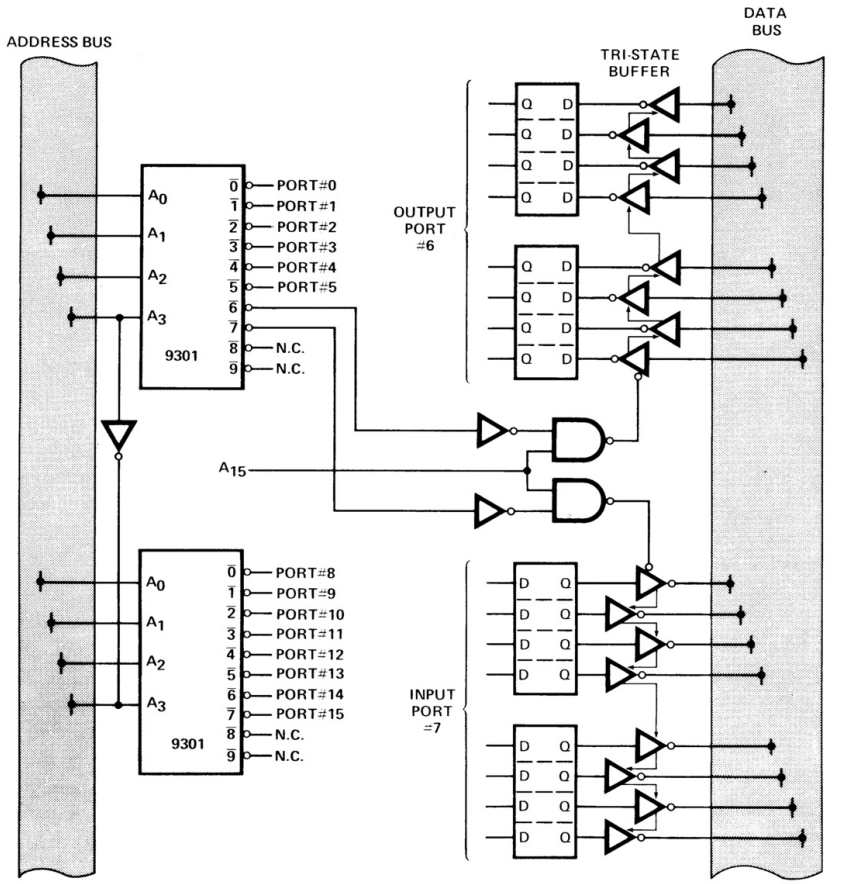


FIG. 3-3. Decoding circuits for 16 memory mapped I/O ports.

INR	Increment Register or Memory
DCR	Decrement Register or Memory
MOV M,R	Move Register to Memory
MOV R,M	Move Memory to Register
ADD	Add Register or Memory to Accumulator
ADC	Add Register or Memory to Accumulator with carry
SUB	Subtract Register or Memory from Accumulator
SBB	Subtract Register or Memory from Accumulator with Borrow
ANA	Logical AND Register or Memory with Accumulator
XRA	Logical Exclusive – OR Register or Memory with Accumulator
ORA	Logical OR Register or Memory with Accumulator
CMP	Compare Register or Memory with Accumulator
MVI	Move Immediate Data to Register or Memory
STA	Store Accumulator Direct in Memory
LDA	Load Accumulator Direct in Memory
SHLD	Store H and L Direct in Memory
LHLD	Load H and L Direct from Memory

FIG. 3-4. The 8080 instructions that can be used to communicate with the I/O ports.

requirements of the 8080. As soon as more than one input source need be capable of interrupting the system things become far more complicated. In general there are three varieties of input port structures of varying complexities and capabilities.

SYNCRONIZED DATA INPUT PORTS

An example of a data input port not using interrupt logic is shown in Fig 3-5. This port is designed to transfer blocks of data from a peripheral device into the microcomputer memory. To facilitate this transfer two handshaking I/O ports are set up. When the system is ready to accept data from the peripheral it sends a logic 1 out on the handshaking output to notify the peripheral to begin sending data. After the transfer of the entire data block has been completed, the handshaking output will be returned to a logic 0 and the transfer will cease.

Although the handshaking output is treated as an 8-bit Output port by the I/O port selection logic, only one bit, bit 7, is picked off the data bus. After the peripheral receives its enabling signal from the handshaking output it transfers the first byte of data to the input port. When the data is correctly latched into the port, the peripheral notifies the system by dropping the handshaking input low briefly and then returning it to a logic high. A possible elaboration on the system shown here is to add a second handshaking output to notify the peripheral when the system has read the data byte and is ready for the next byte. This is desirable if a large amount of processing will be done to each data byte and the possibility exists that the system will not be ready to receive the next byte when the peripheral presents it to the input port. Another alternative is to have this second handshaking output generated automatically by the logic when the data input port is read.

A sample program for reading blocks of data into the memory from the port in Fig. 3-5 is shown in Fig. 3-6. It utilizes the D and E registers to hold the address of the I/O ports, the B and C registers to hold the byte counter and the H and L registers to hold the starting address of the memory block to which the data transfer will be made. Each time a byte of data is transferred, the byte counter will be decremented and the memory address will be incremented. When the byte counter reaches zero, the transfer is complete.

Assuming that the data transfer rate is limited by the speed of the microcomputer and not the peripheral, each of the two loops will be executed only once. Testing for the presence of the handshaking signal requires 28 clock periods or 13.6 μs (assuming a 480 ns clock). The

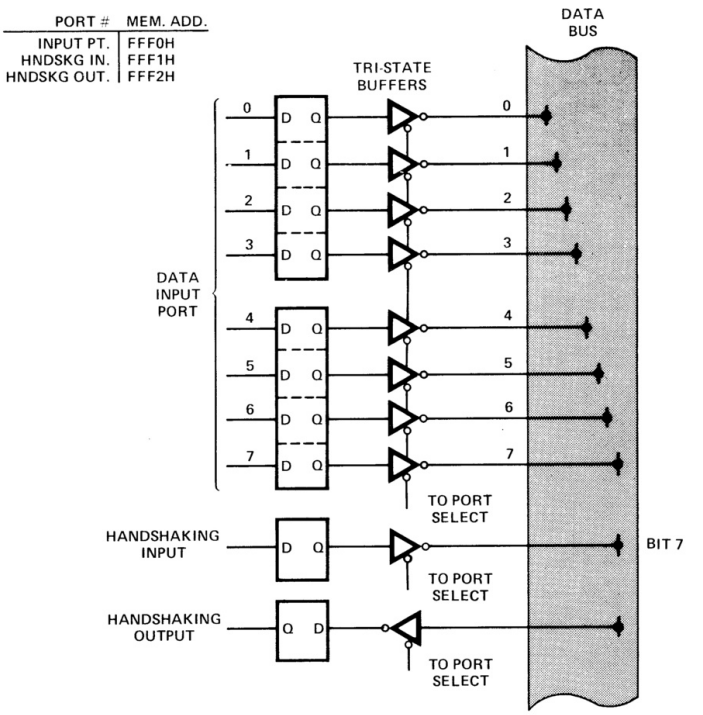


FIG. 3-5. Input port for synchronized data transfer using Hand-Shaking logic.

actual transfer of the data byte requires an additional 46 clock periods or 22.1 μs. Thus each transfer requires a total of 35.7 μs for a transfer rate of 28K bytes/sec.

By adding a slight amount of logic the transfer rate can be increased dramatically. This is done by using the handshaking logic to generate a NOT READY signal which causes the CPU to idle until the signal is removed. This removes the necessity for having the program test for the presence of a signal from the peripheral. The data transfer is initiated as before by sending a 1 to the handshaking output port in bit position #7. The logic automatically generates the NOT READY signal causing the CPU to idle in the WAIT state until the peripheral has the first byte ready to transfer. The handshaking signal from the peripheral causes the NOT READY to be removed from the CPU and the data is read. While the exact data transfer rate is dependent on how accurately the processing speeds of the system and the peripheral can be matched, it is reasonable to expect processing speeds up to one byte every 25 μs or a rate of up to 40K bytes/second using this technique.

POLLED INTERRUPTS

If we are not pressed for high speed data transfers of large blocks of data, the interrupting method may be the best. This technique allows the system to process one program and still interface with another peripheral like a keyboard. One configuration using this technique is shown in Fig. 3-7. The circuit shown is suitable for handling up to eight slow speed peripherals. With slightly more logic more peripherals can be handled easily.

Each peripheral presents its data to the appropriate input port which latches it into the type D flip-flops that make up the port. Thus the latches will always contain the information coming from the peripheral. The data does not find its way onto the data bus until the port select logic enables the tri-state buffers that connect the latches to the bus. When the peripheral has loaded the latches with a byte of data, it pulls the strobe line low. This causes the output of the eight-input NAND gate to go high which causes an interrupt to be fed to the 8080. Notice that any of the eight strobes from the eight peripherals causes an interrupt. When an interrupt occurs the 8080 reads the data bus which will now contain all ones by virtue of the pull-up resistors on each line. This will cause the CPU to read a RESTART instruction and jump to the location specified by that instruction.

The memory location associated with the RESTART instruction FFH is 0038H. Each of the strobe signals will cause the system to jump to that point in the memory, and it is here that the first instruction of the interrupt service routine must be placed. It is important to remember that the interrupt service routine is treated as a subroutine by the CPU and that it is up to the programmer to insure that the working registers are saved in the stack memory. Once this has been accomplished the CPU, under program control, reads the strobe input port to determine which of the eight input ports caused the interrupt. This is easily done since each of the strobes is latched into the D-type flip-flops at the time of the interrupt. Once the address of the interrupting port has been determined it is used to read data into the system from that port.

The program for reading data from the polled interrupt input ports is shown in Fig. 3-8. Very basically the program operates by reading the strobe input port into the accumulator. Since this occurs very quickly after the interrupt occurs there will probably be only one bit in the port since there has only been one strobe signal. We will see in a moment that even if there are several strobe bits in the input port they will be read and serviced. In any event, the bit position of the strobe bit can be determined by rotating the contents of the accumulator to the right

BLOCK

```
; THIS IS A PROGRAM FOR TRANSFERRING BLOCKS OF
; DATA FROM A PERIPHERAL DEVICE THROUGH INPUT
; PORT FFFØH.

        LXI     B, BCNTR    ; LOAD B AND C WITH NUMBER
        LXI     D, ØFF2H    ; OF BYTES TO BE TRANSFERRED
        LXI     H, SADDR    ; AND D AND E WITH ADDRESS
                            ; OF HANDSHAKING OUTPUT PORT. LOAD
                            ; H AND L WITH STARTING ADDRESS
                            ; IN MEMORY WHERE THE DATA
                            ; WILL BE LOADED.
        MVI     A, 80H      ; LOAD 1 INTO BIT 7 OF ACCUM.
        XCHG                ; TRANSFER THIS BIT TO HND SHKG
        MOV     M, A        ; OUTPUT TO ENABLE PERIPHERAL.
                            ;
        DCX     H           ; DECREMENT H AND L PAIR SO
                            ; THAT IT CONTAINS ADDRESS OF
LOOP1:  ANA     M           ; HANDSHAKING INPUT PORT
        JNZ     LOOP1       ; ØFF1H. LOAD INPUT PORT
                            ; INTO ACCUM. WITH ANA M.
                            ; THIS SETS FLAGS. SYSTEM LOOPS
                            ; ON LOOP1 UNTIL PERIPHERAL
                            ; LOADS O INTO PORT. EXIT LOOP1.
LOOP2:  ORA     M           ; LOOP2 TESTS FOR A RETURN TO
        HZ      LOOP2       ; 1.

        DCX     H           ; DCX H CAUSES H, L TO CONTAIN
        MOV     A, M        ; ADDRESS OF DATA INPUT PORT.
        STAX    D           ; MOVE DATA TO ACCUM. AND THEN
                            ; TO MEMORY BLOCK ADDRESSED
                            ; BY D, E.

        INX     H           ; INX H RESTORES HNDSHAKING INPUT
        DCX     D           ; PORT ADDRESS TO H, L. DECRE-
        MOV     A, B        ; MENT BYTE COUNTER AND TEST
        ORA     C           ; FOR ZERO. IF NOT ZERO
        INZ     LOOP1       ; RETURN TO LOOP1 FOR NEXT
                            ; BYTE.
```

FIG. 3-6. Program for synchronized transfer of block of data.

through the carry bit. If we rotate one bit position and then test the carry bit for a one content, the number of RARs necessary to bring the logic one into the carry bit position represents the bit position of the original strobe signal. Thus, it will take four RARs to bring a bit in the fourth bit position into the carry bit. The RARs are executed in LOOP1 and each loop execution results in the incrementing of register D which serves as a counter. When LOOP1 is exited, register D will contain a number that represents the bit position of the original strobe signal. If, for instance, D contains four, we know that it was input port #4 that caused the original interrupt. We will now use this information to cause a jump to the service program for servicing the input port causing the original interrupt.

This program makes use of a jump table to cause the jump to the appropriate service routine. The jump table consists of a set of jump instructions, each to a seperate subprogram for handling a different input port. Each of the programs may be different so that data arriving on port #2 may be handled in a completely different way than the data arriving at port #1. To use the jump table we load the H and L registers with the address of the first byte of the first JMP instruction in the table. Then we add a number to the H and L registers. This number is three times the number representing the bit position (the three comes into the program because each of the JMP instructions in the table occupies three bytes). When this is done the H and L registers contain the address of the JMP instruction appropriate to the interrupting input port. A PCHL instruction causes program control to jump to the address in the H and L registers. This in turn causes a jump to the subprogram to service the interrupting input port.

After the data has been processed according to the individual service routine for that port, a bit is sent to the correct bit position of the control output port. When the peripheral attached to the input port causing the interrupt receives this signal it understands that the byte has been read and processed and it can begin assembling the next byte of data. It is this control signal that keeps the peripheral from trying to load another byte into its input port and swamping the system. When the peripheral receives its control signal it removes the strobe output causing the interrupt.

The subprogram concludes by restoring the working registers from the stack and enabling the interrupts. Because an internal flip-flop in the 8080 is set whenever it receives an interrupt, the 8080 will not respond to further interrupts until this flip-flop has been reset with an EI instruction (Enable Interrupts). We save this instruction until the end of the subprogram so that our interrupt subprogram will not itself be interrupted. Immediately after the EI we execute a RET which takes us back to the original program. If another strobe has occurred on one of the other strobe lines it will cause an immediate interrupt and a return to the POLLD program. Thus we will eventually service each of the interrupts even if they occur while the system is executing the POLLD program.

PRIORITY MASKING

The polled interrupt configuration illustrated in Fig. 3-7 assumed that each of

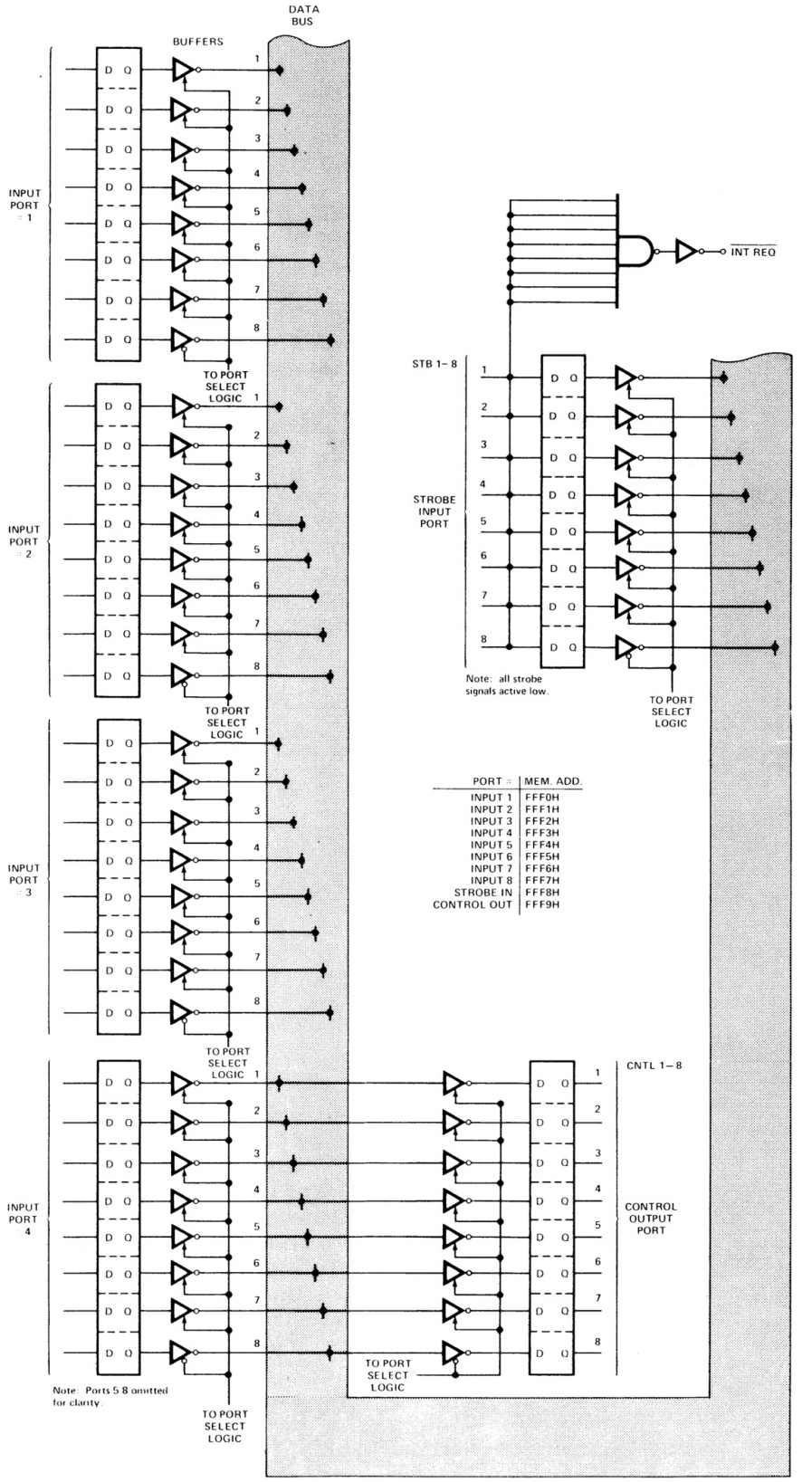


FIG. 3-7. Interrupting Input port using polled interrupts.

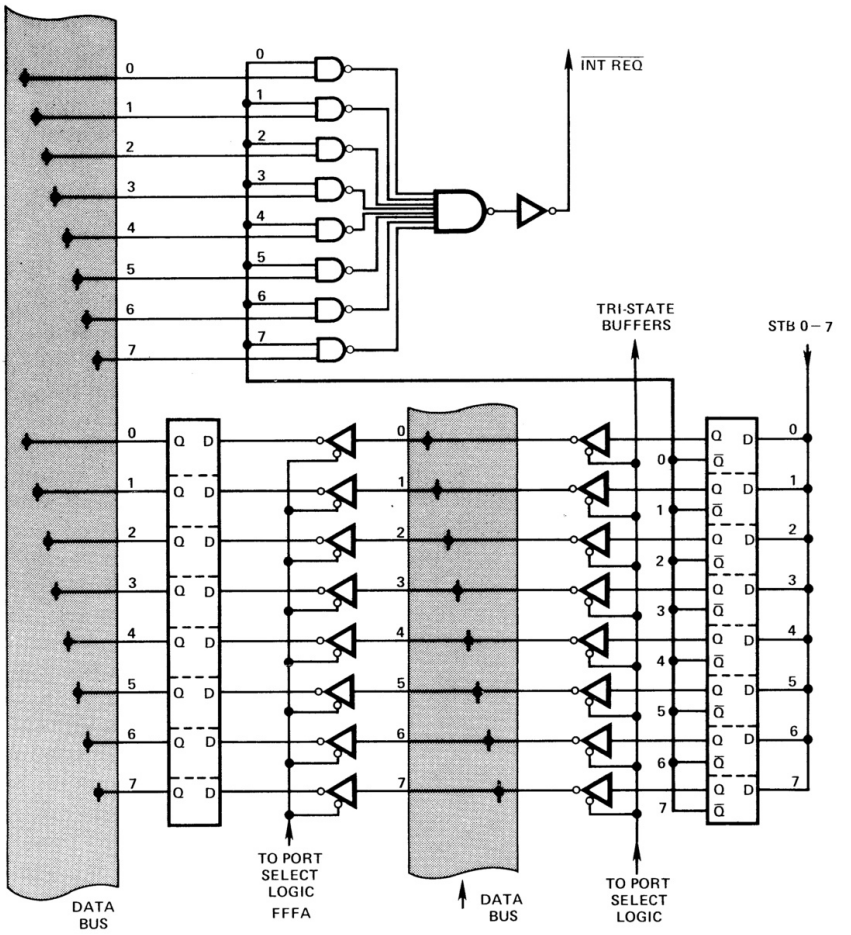
```
                              POLLD

; THIS IS A PROGRAM FOR SERVICING EIGHT
; INTERRUPTING INPUT PORTS BY POLLING
; A STROBE INPUT PORT TO DETERMINE WHICH
; PORT CAUSED THE INTERRUPT.

POLLD:  PUSH B          ; SAVE THE WORKING
        PUSH D          ; REGISTERS AND CPU
        PUSH H          ; STATUS BY PUSHING
        PUSH PSW        ; ONTO STACK.

        LXI B, OFFF8H   ; LOAD ACCUM WITH
        LDAX B          ; CONTENTS OF STROBE
                        ; INPUT PORT.

        MUI D, O        ; LOAD D WITH ZERO

        STC             ; MAKE SURE CARRY
                        ; IS ONE.

LOOP1:  RAR             ; ROTATE ACCUM THROUGH
        INC D           ; CARRY, TESTING FOR ZERO
        JC LOOP1        ; IN CARRY AND INCRMTNG
                        ; D EACH TIME.

        LXI H, STABL    ; LOAD H AND L WITH
        LXI B, 3        ; STARTING ADDRESS OF
                        ; JUMP TABLE. LOAD B
                        ; AND E WITH 3.

LOOP2:  DAD B           ; ADD 3 TO H AND L. DECREMENT
        DCRD            ; D EACH TIME. WHEN THIS
        JNZ LOOP2       ; LOOP IS EXITED D IS ZERO AND
        PCHL            ; H AND L CONTAIN ADDRESS
                        ; IN JUMP TABLE CONTAINING
                        ; JUMP TO THE CORRECT SERVICE
                        ; ROUTINE. A JUMP TO THIS
                        ; POINT IS EXECUTED.

STABL:  JMP ONE         ; PROGRAM CONTROL JUMPS TO
        JMP TWO         ; THE JUMP TABLE LONG ENOUGH
        JMP THREE       ; TO PICK UP THE INITIAL ADDRESS
        JMP FOUR        ; OF THE CORRECT SERVICE
        JMP FIVE        ; PROGRAM.
        JMP SIX         ;
        JMP SEVEN       ;
        JMP EIGHT       ;

ONE:    LXI H, OFFF0H   ; THIS IS THE SERVICE PROGRAM FOR
        MOV A, M        ; READING INPUT PORT #1. IT MOVES
                        ; THE DATA INTO THE ACCUM AND
                        ; PROCESSES IT. PROGRAMS TWO
        LXI H, OFFF9H   ; THROUGH EIGHT ARE OMITTED
        MVI M, 1        ; FOR BREVITY. NOTE THAT EACH
        JMP RESTR       ; MAY PROCESS DIFFERENTLY.

RESTR:  POP PSW         ; RESTR SERVES TO RESTORE
        POP H           ; THE WORKING REGISTERS AND
        POP D           ; ENABLE INTERRUPTS. FINALLY
        POP B           ; IT CAUSES A RETURN TO THE
        EI              ; ORIGINAL PROGRAM.
        RET             ;
```

FIG. 3-8. The program for interrogating and servicing the interrupting Input ports.

the eight peripherals was of equal importance. This is frequently not the case, and unless provisions are made to the contrary, important peripherals will be continually interrupted by less important peripherals. This can be very disruptive to the normal processing flow and will probably result in the important peripheral program being executed at a slower rate.

The solution is to add some sort of priority circuitry to the system so that less important peripherals cannot interrupt more important ones. The priority circuits about to be described are the most versatile in that the priority can be changed during the execution of the program. This is done by using an output port to hold a priority mask word which allows or disallows interrupts from the peripherals depending on their relative importance. Each time an interrupt is recognized the priority mask word is rewritten into the output port to reflect the fact that the system is now interfacing to a new peripheral.

The configuration is illustrated in Fig. 3-9. It uses a set of type-D flip-flops to latch the interrupt requests from the peripherals. These latches form an input port which can be read by the CPU to determine which of the peripheral data input ports is requesting an interrupt. The strobe signals from the peripherals are active low meaning a logic zero represents a strobe or an interrupt request. The Q outputs of the flip-flops are gated onto the INT REQ line so that any one of them will cause an interrupt. The strobes can be overridden by the outputs of the priority mask output port. A zero written into any of these latches will inhibit the corresponding strobe from causing an interrupt. The program for the configuration is shown in Fig. 3-10. It is very similar to the program of Fig. 3-8 except for the addition of the priority mask output at the beginning of each of the service programs. Notice that the EI instruction occurs early in these programs whereas before it occurred at the conclusion of the program. This is because the priority circuits now lock out unwanted interrupts, but higher priority strobes must be free to cause interrupts. Notice also that the priority mask is different in each of the service routines (it is assumed that priorities are assigned in the order of strobe # so that strobe #3 is a higher priority than strobe #2). Finally, it is possible to modify the priorities simply by rewriting the masks in the service programs.



NOTE TO READER

In systems of this type, the large numbers of buses makes the more complex drawings hopelessly confusing. Accordingly, we have adopted a convention of showing the subsidiary buses as single lines. Inputs and outputs are numbered to allow the reader to match up the circuits as they feed off the bus.

FIG. 3-9. Priority masking added to the polled interrupt port structure of Fig. 3-7.

HARDWARE PRIORITY INTERRUPTS

While the priority masking scheme illustrated in Fig. 3-9 is versatile, it is not particularly fast since the routine for determining which port caused the interrupt consists of 15 or so instructions which must be re-executed for each interrupt. This is certainly a modest program load for slow peripherals but for the utmost in speed it is possible to improve on the performance of this circuit by modifying the hardware.

The new circuit, shown in Fig. 3-11, is faster than the polled method of port interrogation but suffers from the disadvantage of having the priorities permanently assigned by which inputs they are tied to. A further disadvantage is in the number of RST locations the circuit uses.

The 8080 has a total of eight RST instructions in its instruction set. These are normally used with multiple interrupts so that each interrupt causes a jump to a different address where the service program for that interrupt is located. This method avoids the use of jump tables and indexed addressing and is therefore faster. The method is only suitable for up to eight interrupting peripherals since there are only eight RST instructions. The polled method of interrupting input port interrogation works with any number of ports since it uses only one RST instruction.

The circuit of Fig. 3-11 uses an eight-input priority encoder, the 9318, to generate the machine code for the eight RST instructions. The machine code for RST is 11XXX111, where XXX is a binary number that determines the location the jump will be made to. Since all of the bit values of RST are logic ones, with the exception of XXX, these can be assured by placing pullup resistors on the data bus. The three outputs of the 9318 provide the value for XXX depending upon which of the eight strobe lines has gone low. Any of the strobes cause the GS signal to go low which provides an interrupt request to the 8080. When the CPU responds to the interrupt by sending out the INTA signal, this signal is used to gate the XXX outputs of the 9318 onto the bus. The result is a jump to the appropriate service routine.

The circuit of Fig. 3-11 is simplified considerably. In normal use it is necessary to provide latches for each of the input data ports as well as the strobe signals. In addition, provision must be made for inhibiting or enabling selected strobe signals from causing interrupts. Because of the marginal improvement in processing speed of the hardware priority circuits over the polled method, we will not consider these circuits further. Interested readers are referred to the Iasis ia515 Programmed Learning Course, Vol. V, for additional material.

```
POLD 2:  PUSH B
PUSH     PUSH D
         PUSH H
         PUSH PSW
         LXI B, 0FFF8H
         LDAX B
         MVI D,0
         STC
LOOP 1:  RAL
         INC D
         JC LOOP1
         LXI H, STABL
         LXI B,3
LOOP 2:  DAD B
         DCR B
         JNZ LOOP2
         PCHL
STABL:   JMP ONE
         JMP TWO
         JMP THRE
         JMP FOUR
         JMP FIVE
         JMP SIX
         JMP SEVEN
         JMP EIGHT
RSTR:    POP PSW
         POP H
         POP D
         POP B
         RET
ONE:     LXI H, 0FEH
         EI
         LXI H, 0FFF0H
         MOV A, M
          .
          .
          .
         LXI H, 0FFF9H
         MVI M, 1
         JMP RESTR
TWO:     LXI H, 0FFFAH
         MVI M, 0FCH
         LXI H, 0FFF1H
         MOV A, M
          .
          .
          .
         LXI H, 0FFF9H
         MVI M, 2
         JMP RESTR
```

FIG. 3-10. Polled II.

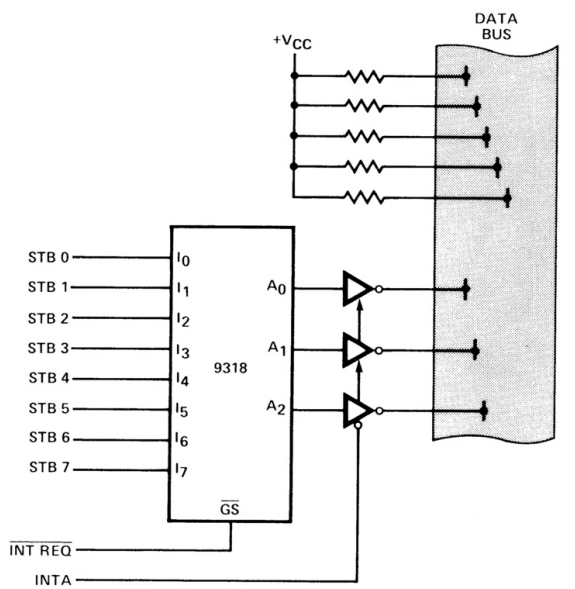


FIG. 3-11.

Chapter Four
The Role of the Development System

To fully appreciate the role of the development system in the task of programming and debugging a working microcomputer system, it is best to examine the process of writing a program. Consider for a moment, the problem of programming the world's first microcomputer system. All microcomputer systems operate by setting their internal program counter to zero when they are first energized. The contents of the program counter are then sent out on the address bus to the memory to fetch the first instruction. Of course, the world's first system will not have an instruction in the first location since there is no program in memory!

Here is the crux of the problem. How is the program loaded into the memory? Programs consist of instructions that are in turn made up of one or more bytes of binary data which are then stored in memory locations. An examination of a memory containing a program will not reveal whether any particular location contains data or an instruction. The CPU proceeds on the basis that unless specifically told otherwise, every memory location contains an instruction and not data.

Instructions are loaded into the memory by presenting them to an input port followed by an address. The instruction is moved to the accumulator, the address is moved, byte by byte to the H and L regsiters, and the instruction is then moved to the appropriate location in memory using the H and L registers as a memory pointer. See Figure 4-1. Thus instructions are moved into memory through an input port exactly like a data word.

The program of Figure 4-1 is simplified somewhat in that no provision for handshaking signals has been made, nor was a byte counter set up. Even so, it is obvious that a special program is required for loading instructions into memory. Of course, if this is truly the world's first microcomputer system, how was the program for loading instructions itself loaded into memory? After all, it too is made up of instructions that somehow must be loaded into memory locations. The problem is solved in the world's first system by loading the memory using DMA activity to bypass the CPU and load the memory locations directly. The program for loading instructions into the memory is called the bootstrap loader. Once it has been loaded into the memory with DMA, other programs can be entered through the input ports.

Unfortunately, the bootstrap loader program will disappear from the memory every time the system is turned off. Each morning the bootstrap loader will have to be manually keyed into the memory using DMA techniques. Besides the problem of entering instructions into memory, the user trying to write and debug programs will want to be able to do other things. Suppose he finds it necessary to insert one instruction in the middle of the program. That requires that each instruction after the instruction be moved a few bytes further into the memory. This feature requires a further elaboration of the bootstrap loader.

To key in the bootstrap loader it was necessay to work with switches, one switch per bit. Trying to enter a long program this way would take forever. It would be very nice if the computer system understood ASCII coded characters so that the user could interface with the computer via a keyboard like that found on an ASR-33 Teletype machine. This has the further advantage of producing a typed copy of the program as it is written by the user. If the teletype is equipped with a paper tape reader and punch, programs can be saved on paper tape. This allows multiple users to work on their programs more or less simultaneously in that one user can be debugging a program listing on paper while his partner is running his program on the system. It is a fact of life for programmers that programs never run correctly the first time so that provisions for allowing multiple users to interface to the machine are very desirable features. Of course, the paper tape interface requires its own program.

When all of these little programs for interfacing to the system are added up, the resulting master program is called the system monitor program. Real life monitors become very complex and may involve throusands of instructions. Having to key a program of this length into the system each time it was energized would be extremely time consuming and prone to errors.

The solution lies in the construction or purchase of a system dedicated to the

```
        LXI H, 0FFFFH     ; LOAD H AND L WITH
        MOV D, M          ; ADDRESS OF INPUT
        MOV E, M          ; PORT. LOAD
        XCHG              ; STARTING ADDRESS
        LXI B, 0FFFFH     ; INTO THE D AND
LOOP 1  LDAX B            ; E REGISTERS AND
        MOV M, A          ; XCHG WITH H + L.
        INX H             ; READ INSTRUCTION
        JMP LOOP 1        ; AT INPUT PORT
                          ; AND MOVE TO
                          ; MEMORY. INX H
                          ; AND L, GET NEXT
                          ; INSTRUCTION AND
                          ; REPEAT.
```

FIG. 4-1 A simplified program for moving instructions and addresses presented to the system at input port FFFFH into memory.

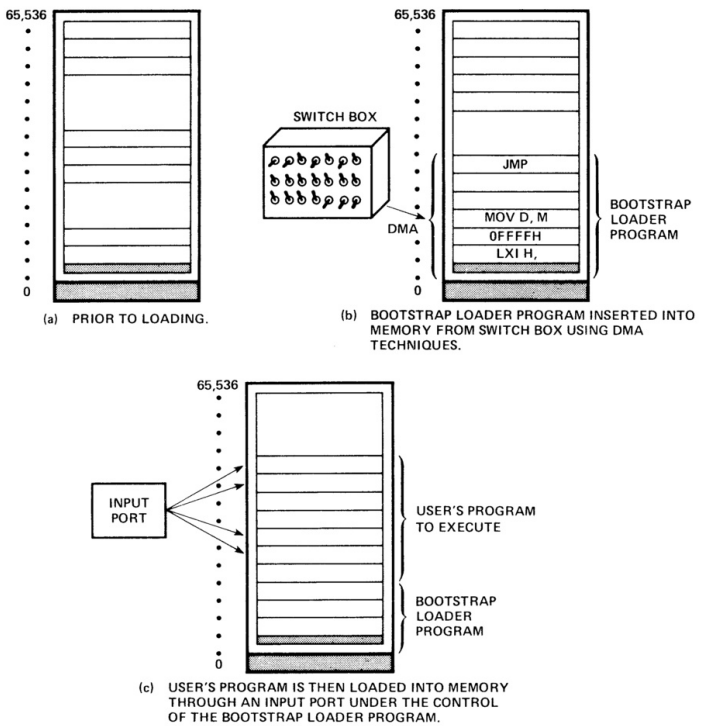


FIG. 4-2 Once the Bootstrap Loader Program has been inserted into the memory, it can be used to load other, longer programs.

development of microcomputer systems. The system monitor program is supplied with the system in the form of ROMs so that the monitor is instantly available when the system is energized. A good system will make it as painless as possible to write and debug microcomputer programs. This is accomplished with both software and hardware aspects of the design.

DEVELOPMENT vs. OEM SYSTEMS

While the development system is dedicated to the application of writing and debugging programs, it should be noted that it is itself a complete microcomputer system. Thus a development system designed to write and debug programs for an 8080 microprocessor will itself contain an 8080. Microprocessor systems are pretty basic so that functionally the development system will not be much different from the system it is simulating. Both will contain a CPU, some memory, I/O ports, timing and control signals, etc. The differences are in type and quantity of memory and the use of I/O ports.

It should be obvious now that microcomputer systems must fulfill several roles in the life cycle of a typical product. Initially a specialized development system is used for writing and debugging the program. Then the OEM system will be simulated so that the program and I/O ports can be tested in a working environment. For this purpose the development system will probably be called upon. Next a production prototype will be needed. In this phase it is important that the system resembles as closely as possible the production OEM system. In fact it will probably only differ in the use of PROMs instead of ROMs and the extensive use of sockets for the ICs. The production prototypes will be assembled into the user's system and tested for proper opration in adverse environments such as high electrical noise situations.

In the early days of microcomputers, program development was done on development systems that bore little resemblance to the OEM microcomputer systems that went into production as the final product. The development systems were made up of large, complex printed circuit boards containing a great deal of circuitry for performing the specialized tasks of writing and debugging programs. These boards were initially offered for sale as OEM boards, but the large amount of circuitry on the board dedicated to program writing and hence, redundant, made them a very inefficient choice for OEM usage.

Most users preferred to develop their own OEM systems using the development system only for program development and programming PROMs. This was economically attractive only in applications requiring 100–500 systems, where the number of systems was smaller, the use of microcomputers was awkward to say the least.

Eventually, OEM oriented systems began to appear on the market. Initially these were multiboard systems optimized for large, complex applications. Prices ranged from several thousand dollars down to one thousand dollars. Recently

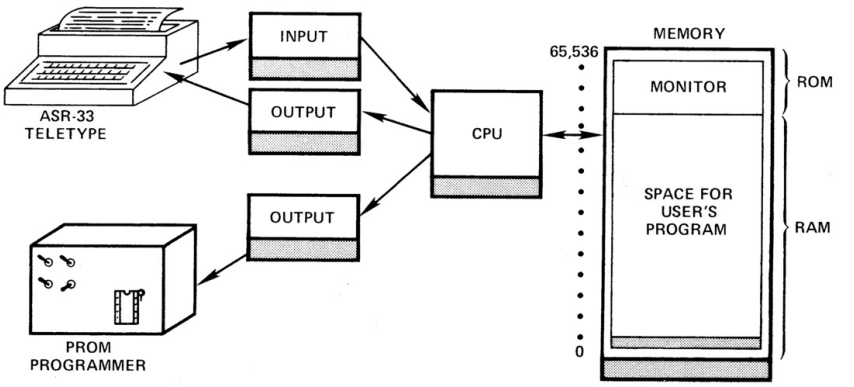


FIG. 4-3 A typical development system. Since the user will naturally wish to begin his program at location 0, the Monitor Program is placed at the top of the memory and some logic circuits added so that the system, when first energized, will immediately jump to the monitor. Programs are written on the teletype and loaded into the memory under the direction of the monitor. If they execute correctly, the program can be loaded into PROMs, also under the direction of the monitor.

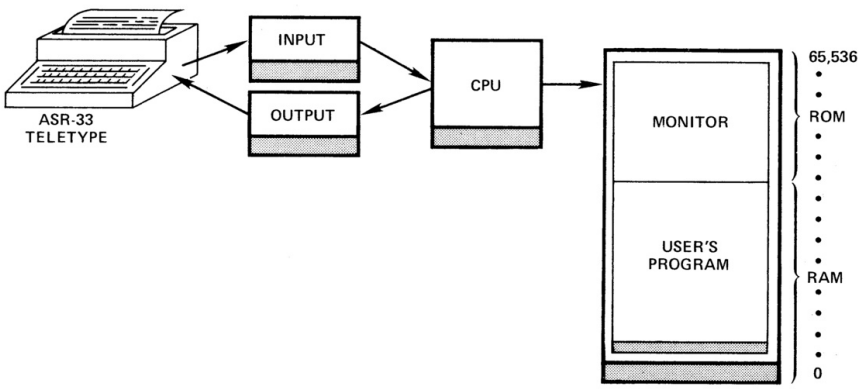


FIG. 4-4 The program is written on the teletype and loaded into the RAM memory. It normally begins at location 0.

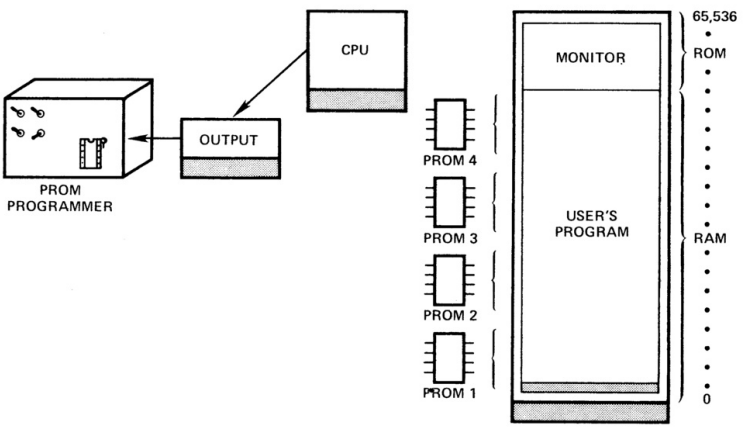


FIG. 4-5 Once the program executes correctly, it is output through the PROM Programmer into a set of PROMs.

single board systems have appeared on the market for as little as $500. Unfortunately the multiboard systems were not designed to be compatible with the development systems, so that the transfer from development stage into production was awkward. Even the single board systems, which are manufactured by the producers of the development systems, have had architecture differences and have made little or no use of techniques that could have made the design transfer process painless. The design goals for the development system, versatility and ease of use, and the design goals of the OEM system, mainly low cost, have led these two systems down diverging paths.

It is truly unfortunate that this philosophy has prevailed, since it is possible to build compatible OEM and development systems that are simultaneously versatile, easy to use, and cost effective.

THE VERY REAL NEED FOR COMPATIBLE OEM AND DEVELOPMENT SYSTEMS

Perhaps the best way to justify the need for compatibility in OEM and development systems is to simply list some areas of potential compatibility and the advantages that accrue to the user of a system so configured.

One of the biggest headaches for microcomputer system developers is that of debugging systems once they have reached the production prototype stage. For most developers this stage consists of loading up a set of PROMs or EROMs with the program that has been debugged in the RAM section of the development system. The PROMs are plugged into sockets in the prototype board and the system energized. This represents a first for the system in two regards. It is the first time the microcomputer system has operated in anything like the real environment the system will see in practice, and it is probably the first time the user has tried driving dedicated I/O devices like relays and triacs with his program. He now has many exciting hours ahead of him tracking down noise interference and trying to determine if his problems are caused by noise, poor I/O design, program glitches or problems associated with the host system.

A properly designed OEM/Development system combination will break this stage into two separate stages. First, the prototype board will be run using its CPU clock, control circuits and I/O ports, but using the memory of the development system. This saves having to erase and reload EROMs each time a program change is made (and there will be many

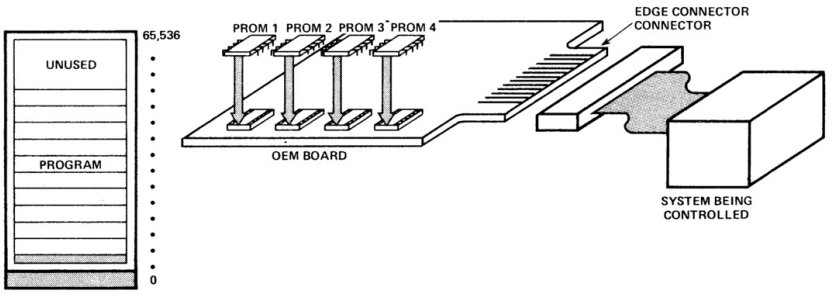


FIG. 4-6 The PROMs programmed by the development system are plugged into sockets on the OEM board. The resulting system represents a production prototype. Not all of the memory can be used — that portion occupied by the monitor in the development system must be vacant.

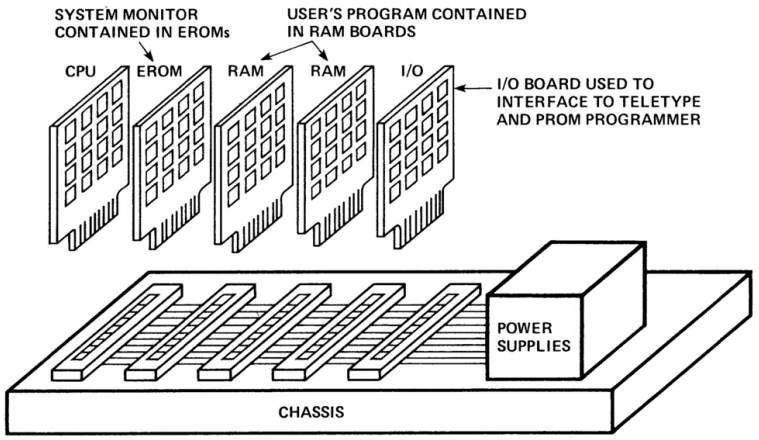


FIG. 4-7 A typical dual-purpose multi-board system used in the development system mode.

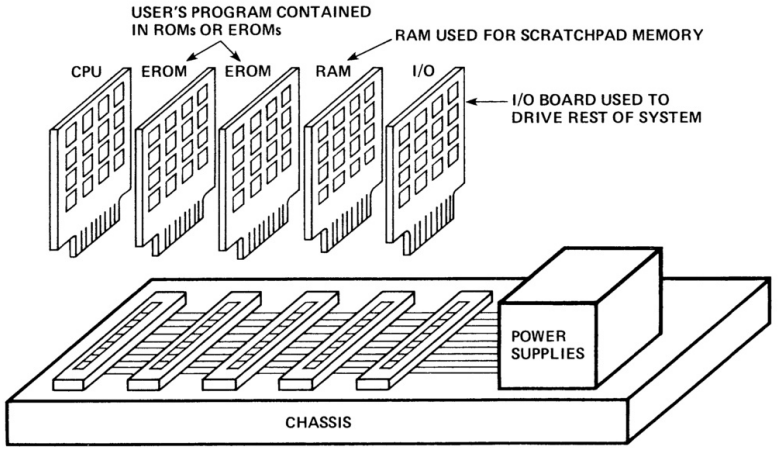


FIG. 4-8 The dual-purpose system used in the OEM mode.

changes). Instead, changes in the program can be made as simply as they were during the initial program writing phase. In the second stage the EROMs will be loaded with the program and the system run using its own memory. At this point the problems should be pretty well worked out.

A properly designed OEM system will make provisions for debugging even at this later stage. There are several ways to do this. One involves the use of null instructions. An examination of the 8080 instruction set reveals that some very valid instructions are omitted from listings of the instruction set because they are considered trivial. The MOV instructions fall into this class. They are normally used in the MOV A,B form where B is the source register and A is the destination register. Not listed, though just as valid, are MOV A,A, MOV B,B, MOV C,C, etc. These are null instructions because they perform no useful action; i.e., MOV A,A accomplishes nothing but replace the contents of the accumulator with the contents of the accumulator. However, the CPU actually executes this instruction in the fashion just described.

If the OEM system has the necessary logic circuits, these null instructions can be detected and made available on the system bus. One very useful way to use the null instructions is as program breakpoints. If the development system is so equipped, switches can be set on the front panel so that when the OEM system executes one of the null instructions the system is halted. Now the user can drive the system in a single step mode examining the registers at every step until the glitch is located and corrected.

This feature is not normally used at the time of the program development which is done on the development system and there are simpler ways of setting breakpoints. There are, however, problems frequently associated with bringing the system up in the application itself. Many times this is done on location in a remote spot far away from the development system. A technician armed with a simple debugging system utilizing the null instruction can install the system if breakpoints have been inserted in strategic points within the program. This feature can be a tremendous aid in working with the OEM system, yet it is normally included only in development systems. This is a perfect example of the potential benefits to a compatible OEM/development system. This topic will be explored in greater depth in later chapters.

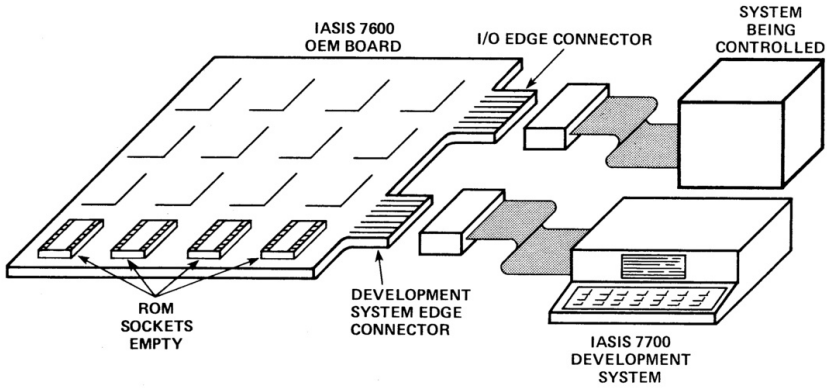


FIG. 4-9 When the Iasis ia7600 OEM Board is used in conjunction with the ia7700 development system, the 7600 CPU, clock, control and I/O circuits are used to drive the system being controlled. The memory of the 7600 is disabled and the program executed out of the 7700 memory. This allows the program to be displayed on the CRT of the 7700 and quickly changed without going through the PROM stage.

MACHINE LANGUAGE PROGRAMMING AND ITS DISADVANTAGES

Early microcomputer users were forced to write and enter their programs in machine language. In this method the program is written out on paper using any convenient set of mnemonics for the instructions. The binary code or the hexadecimal code of each of the instructions is then written alongside the mnemonic of each of the instructions in the program. These codes, representing the program in machine language, are then entered into the memory through one of the input ports and executed.

There are two very real disadvantages to this approach. The first is the possibility of making an error in the laborious process of converting each of the instructions in the program into the machine coded equivalents. This possibility is compounded because the program cannot be easily scanned for errors once it has been encoded.

The second disadvantage is more troublesome. The first can, after all, be eliminated by sufficient care and review in the encoding process. The second is present no matter how much care is used in the encoding process. It is associated with the assigning of memory locations to the destinations of jump instructions and other instructions that reference memory locations.

During the process of writing a program the programmer normally discovers that certain program segments appear over and over again. If he repeats the set of instructions representing that segment each and every time the segment is to be used, those instructions will be repeated in memory many times. The programmer learns very quickly that in microcomputers, memory is money. Repeating program segments over and over again is very wasteful of memory and runs the system cost up. If, instead of merely repeating the program sequence, the programmer treats it as a subroutine call, the instructions will only appear in memory once. Now the system executes the subroutine by jumping to the first address of the subroutine which begins the short process of saving the working registers in the stasck memory. The subroutine is then executed; this is, in effect, the repeated program segment. At the conclusion of the subroutine the registers are restored from the stack and program control returned to the main program.

In addition to the savings in memory bits by using subroutines, a further advantage is realized by using conditional call instructions. These fall into the category of branch instructions which the system executed by testing one of the status bits and then either jumping or not jumping dependent upon the state of the status bit. This allows the system to test for sign, parity, carry and auxiliary carry and jump to a subroutine depending upon the state of those conditions. This, in effect, gives the system the ability to make decisions. The 8080 has a complete set of branch instructions for jump, call-to-subroutine and return-from-subroutine, certainly an extremely powerful alternative to the simple expedient of merely repeating the instructions of common program segments.

A closely related problem is that of repeating a set of instructions over and over until some preset condition occurs. This frequently occurs when setting up a counter within the program so that some action will occur n times where n is a variable determined by some other aspect of the program. In practice this is done by executing the program, testing the condition and then executing a conditional branch back to the beginning of the segment if the condition has not yet been met.

The use of these very powerful techniques does present a problem to the programmer and it is here that we experience the real disadvantage to writing programs in machine code. Each of these instructions involves a memory reference, that is an instruction that specifies another memory location. Locations are specified by addresses so that when a program is written in machine code the specific addresses of each of the jump destinations must be filled in. Unfortunately in some cases the programmer does not know all of the addresses until the program is complete. Usually the program is written in

the form of one main program with many jumps and calls to subroutines and other program segments which appear at the end of the main program.

The machine language programmer attacks the problem of writing programs by assigning memory addresses to each of the instructions as they are written, taking care to assign three locations for a jump instruction, one for a NOP, and so forth. Each time a memory reference instruction is reached a temporary symbol such as X1 is inserted in place of the address. Later when the jump destination is reached and that address becomes known the programmer can backtrack and replace the temporary symbols with the actual addresses. Loops are not as difficult since the jump destination address is already known when the jump instruction is encountered.

If this seems like a small burden, consider what must happen when the programmer decides to add or delete an instruction. After all, programming can be considered an art of finding omissions; another way of looking at this is that programs never get simpler, only more complex, so that the programmer continually finds himself inserting instructions into the program. Each insertion requires the programmer to go back and change the addresses of each of the instructions so that the new addresses ripple through the program. He is also forced to go through the program, checking each memory reference instruction to see if it has been affected by the address changes and to modify it accordingly. Needless to say, the chances of making an error during this process are very high.

Since the microcomputer system only understands binary or, at best, hexadecimal, all numbers and addresses that appear in the program must be written in those number systems. This can become very awkward and forces the programmer to keep a set of conversion tables handy. It also makes it difficult to spot errors later on.

ASSEMBLY LANGUAGE PROGRAMMING

Fortunately there is an easy alternative to machine language programming. The task of keeping track of memory addresses while the program is being written and debugged can be taken over by the microcomputer system itself by using a special program called an assembler. There are now three programs floating around our development system, the system monitor permanently contained in ROMs, the assembler program stored on paper tape, and the program that is being debugged which the programmer has written on a piece of paper. The program undergoing debugging is called the source program.

Writing source programs in assembly language is going to remove a large portion of the housekeeping task from the programmer and allow him to use his time for the purely creative work of programming. However, assembly language programming does introduce a few restrictions of its own. Lest the reader jump to the conclusion that these restrictions negate the advantages of assembly language programming, let us point out that the assembly language rules or syntax are easily learned and will be committed to memory quickly by any serious programmer, so this does not represent a burden to him.

Programs written in assembly language must follow a preestablished format. The page the program is written on will be divided into four fields or spaces. The left-most field is called the label field, the second field is the code field, the third is the operand field and the fourth is the comment field. Most programs today are typed onto a teletype machine so that the four fields exist as imaginary areas on the paper roll in the teletype. Usually the programmer has already written out his program on a piece of paper which acts as a worksheet for him. He then types it onto the teletype which has been set to produce a punched paper tape of the program as it is received. The roll of paper in the carriage of the teletype serves to give the programmer a hard copy of his program so he can scan it for typing errors. The paper tape produced at this time is called the source tape.

Assembly languages require that the programmer use a fixed set of mnemonics that will be recognized by the assembler program. For instance, the Jump If No Carry instruction will always be typed as JNC; JNC is the mnemonic of the instruction. In most published lists of the instructions recognized by a microcomputer, the mnemonics listed with each instruction are those recognized by the assembler program. When the programmer types the program on the teletype he inserts the instruction mnemonic in the code field. That is the only information that can go into this field; anything else will cause an error. Many instructions require additional data besides the mnemonic itself. For example, the MVI instruction, Move Data Immediate, requires that the register receiving the data be specified as well as the data itself. Both the register name and the data are typed into the operand field. Whereas the microcomputer only recognizes binary

data, the assembler program has conversion subroutines built into it so that it will accept octal, decimal or hexadecimal numbers and convert them into binary. It has also preassigned letters to the working registers of the CPU so that it recognizes A as the accumulator, B as register B, etc. Thus MVI B, 114 is recognized by the assembler program as Move Immediate to register B the equivalent of decimal 113. Of course, the assembler has to be able to tell whether the number it is receiving is in binary, octal, decimal or hexadecimal. Accordingly, letter suffixes are added to all of the numbers that appear in assembly language programs. Binary numbers are always followed by B (11010011B), Octal by either an O (oh) or a Q (577Q), Decimal by either a D or no suffix at all (113D or 113), and Hexadecimal with an H (6FA9H). In addition, so that the assembler can separate numbers from character strings (which always start with an alphabetical character), all numbers must begin with a numeral. In particular, hexadecimal numbers like FFFFH or B8F3H must be typed as 0FFFFH and 0B8F3H, respectively.

The comment field is reserved for the programmer to write notes to himself and other readers to explain and annotate the program as it progresses. The information in the comment field does not find its way into memory so that there is every reason for the programmer to write clear and lengthy notes in this field to make the operation of the program clear to everyone who may need to understand the execution.

The label field is exclusively reserved for addresses of each instruction. However, since the assembler performs the operation of assigning the memory locations, this field is not used for the specific number of the address. Instead, symbolic addresses called labels are inserted into this field. Each label must begin with an alphabetic character, a @ or a ? and cannot be more than 5 characters in length. Except for the first character of the string, letters and numerals can be mixed so that LOOP1, for instance, is a valid label. Only some of the instructions will need the symbolic addresses or labels so that the use of this field is optional and left up to the individual programmer.

As we shall see shortly the use of symbolic addresses or labels removes the problem of having to assign specific addresses to each of the instructions. Instead, this task is handled by the microcomputer system itself under the direction of the assembler program. Each of the subroutines and loops in the program are assigned labels by the programmer. These are chosen to correspond to the function of the subroutines so that the programmer can remember the labels easily and facilitate programming. Thus the subroutine to move a block of data from one part of the memory to another might begin with an instruction labeled BLOCK. The assembler program assigns an actual address to the BLOCK subroutine such as 28F7H. When the programmer writes CALL BLOCK the assembler program interprets it as CALL 28F7H. If in the course of debugging the program the user has to insert two instructions the assembler program will correct the address assignments so that CALL BLOCK is now interpreted as CALL 28F9H. Thus, as far as the programmer is concerned, the addition or deletion of instructions no longer affects the address assignments of the instructions in the program.

OPERATION OF THE ASSEMBLER PROGRAM

Once the user's program has been typed onto the teletype and a source tape created, the next step is to load the assembler program into the system memory. Since the assembler program is contained on paper tape, it is loaded into the tape reader of the teletype and read into the system memory. Naturally, this must be done under the direction of the monitor. The assembler program requires a substantial amount of memory — as much as 7000 bytes of RAM for the 8080.

With the assembler program residing in the memory of the system it is now ready to receive the source program. If the assembler program occupies 7000 bytes of RAM, and many systems only have 8000 bytes of RAM available, how is there going to be room in the memory for the source program as well? The answer is there isn't. Instead, the source program will have to be read into the system several times, each time only the information needed at that time is extracted. During the first pass of the source tape the assembler searches for character strings beginning with alphabetic characters. Of these, two are of particular interest: instruction mnemonics and labels. Since each label must begin with an alphabetic character and is followed immediately by a colon to separate it from the code field, the labels are easily identified by the assembler program. The instruction mnemonics are a little more difficult to recognize, but since the assembler program contains a table of all the mnemonics in the instruction set, it can verify the character string by simply

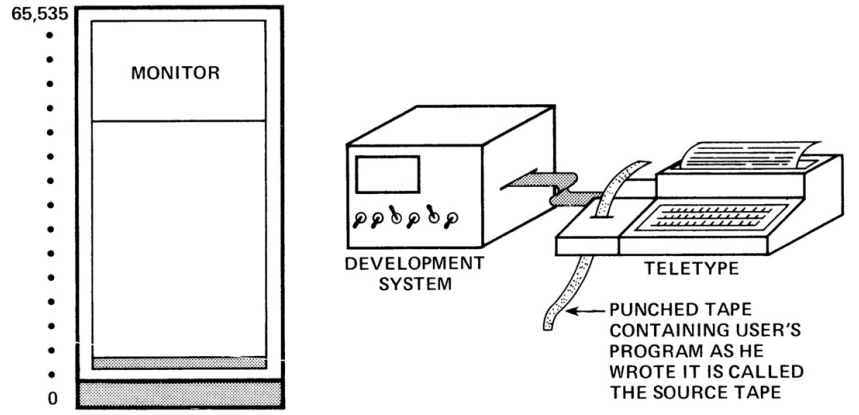


FIG. 4-10 To assemble a program, the user types it in a highly-defined format onto the teletype. He simultaneously punches a tape of the program as it is typed.

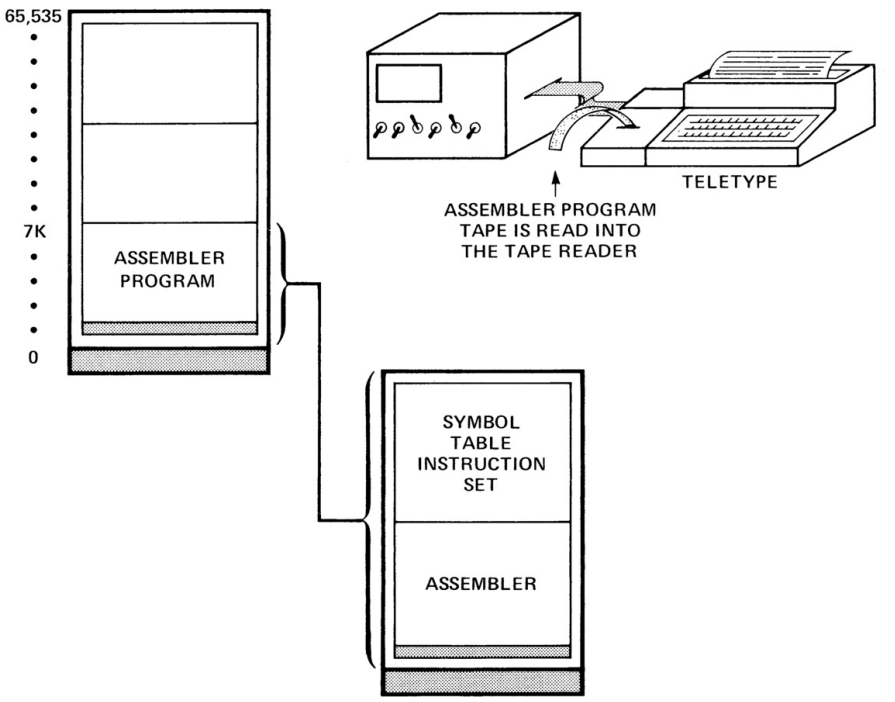


FIG. 4-11 Once the source tape has been created, the assembler program must be loaded into the system memory. This is done under the control of the system monitor. *The first portion of the symbol table is built at this time.* It consists of each of the mnemonics for the instruction, the number of bytes each instruction occupies in memory, and the type of data that must accompany the mnemonic.

checking it against each of the mnemonics in the table.

The assembler program operates by doing two things as the source tape is read in. First it builds a two-part symbol table. Each entry in the instruction portion of the table consists of the instruction mnemonic, the machine code of the instruction, the number of bytes if occupies in memory and some other parameters that will be used in the assembly of the program. This portion of the symbol table is already in the system memory before the source tape is read in. Thus NOP (No Operation) is listed along with the machine code, 0000 0000, and the number of bytes it occupies, one. So is JMP (Jump), the machine, code 1100 0011 and the number of bytes it occupies, three. Also specified in the table is the fact that the two bytes that follow the machine code for JMP is the address representing the jump destination; these two bytes should not be interpreted as instructions.

Each time the assembler finds a label in the label field of the source program, it adds that label to the label portion of the symbol table. Thus the symbol table grows as the source tape is read in. Notice also that the assembler only picks up labels when they appear in the label field. Each label will appear once in the label field and at least once in the operand field of some instruction in the program. After all, if the label is not used as a memory reference in the operand field there was no reason to assign a label to that instruction in the first place. In any event, the assembler ignores labels appearing in the operand field and only uses the labels in the label field to build the symbol table. The reason for this will become apparent shortly.

The assembler program also sets up two memory locations as the location counter. This is a pseudo program counter and is designed to duplicate, under program control, the operation of the program counter as it will occur during the execution of the user's program. Unless specified otherwise, the memory locations are initialized by the assembler as 0000H. When the first instruction of the source tape is read into the system, the assembler program checks each character string to find the labels and mnemonics. If it finds a mnemonic, it checks the symbol table and increments the location counter by the number of bytes that the instruction will occupy in memory. Thus if the first instruction is a *MOV the location counter will be incre*mented by one. If it finds a JMP instruction mnemonic it will increment the loca-

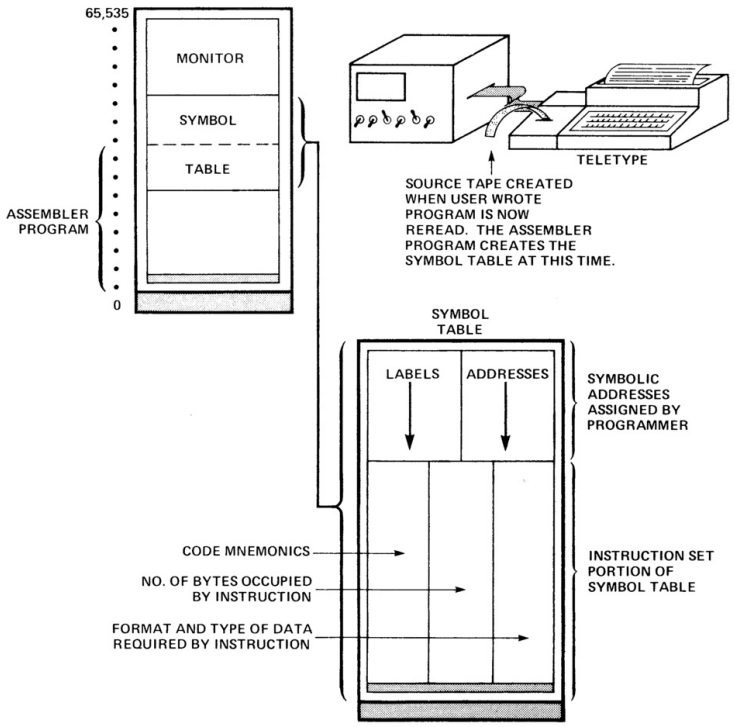


FIG. 4-12 The source tape is now read into the system. The assembler program monitors the tape and checks each character string preceding a colon to see if it is already in the symbol table. If it is, the instruction mnemonic is ignored. If not, it is added to the symbol table as a label.

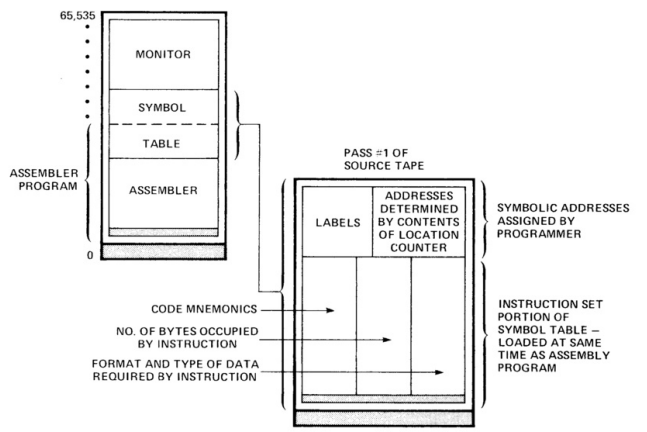


FIG. 4-13 During the first pass of the source tape the symbol table is completed. This is done by setting up a location counter in one of the register pairs of the CPU. As each instruction is read from the source tape the assembler looks it up in the symbol table and increments the location counter by the number of bytes that the instruction occupies in memory. Whenever a label appears it is added to the symbol table along with the contents of the location counter.

tion counter by three. In this way the location counter keeps track of which memory address will contain each instruction of the user's program.

Every time the assembler finds a label in the label field it writes that label into the symbol table as a new symbol along with the cotnents of the location counter at that time. Thus for the label portion of the symbol table the table is made up of entries of labels and addresses. Thus the system will know in the future when that label appears in the operand field as a jump address or a call subroutine what the address of the first instruction of that loop or subroutine is. This is why labels in the operand are ignored, they do not contain address information that can be used in the building of the table.

After the first pass of the source tape the symbol table is completed and the location counter has been used to assign addresses to each of the labels in the source program. The source tape is now reloaded into the tape reader and a second pass made. During the second pass the source program will be read again, but now it will be combined with information in the symbol table and the results printed out on the paper roll of the teletype. The format of that printout is such that the programmer can see whether the program as written has a chance of running correctly. First the address of the instruction is typed in hexadecimal. This is followed by the machine code of the instruction, also in hex, that is to be inserted into that memory location. The address is simply the current contents of the location counter which has been reset to zero prior to the second tape pass. This entry is followed by an error listing which tells the operator if he has made any syntactical errors such as trying to start a label with a numeral. During this second pass the assembler ignores labels in the label field and only responds to labels in the operand field. Every time it finds one the symbol table is searched until the label is located there and the address retrieved that was previously assigned by the location counter during pass one. This address is used in the program in place of the label. Thus the two bytes that appear in the two locations following a JMP instruction are the address bytes from the symbol table corresponding to the label referenced in the JMP instruction. At the conclusion of pass two all of the labels and symbolic addresses have been removed from the program and replaced with the actual numeric addresses. After typing the error listing the assembler types the source program as written by the programmer for a convenient reference.

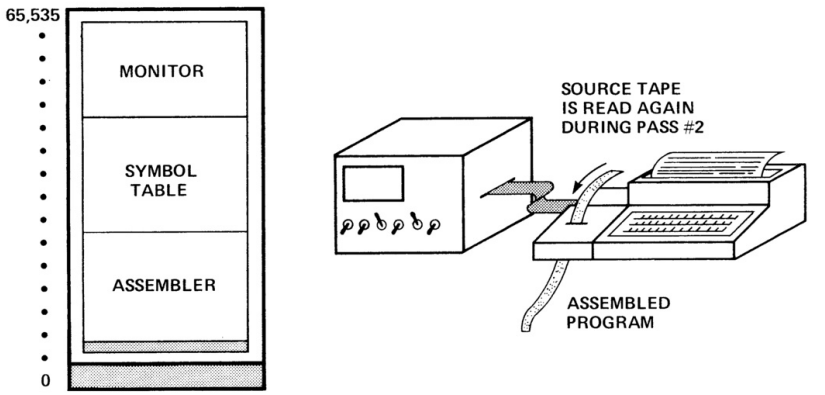


FIG. 4-14 The source tape is read into the system once more. This time the assembler converts the numbers to binary and punches an assembled program tape out. Each time a label is used as a memory reference, like JMP THERE, the symbol table is consulted and the address in the table used to replace the label. The codes or mnemonics are also replaced by their machine coded equivalents. A third pass of the tape allows the teletype to create a hard copy listing of the assembled program.

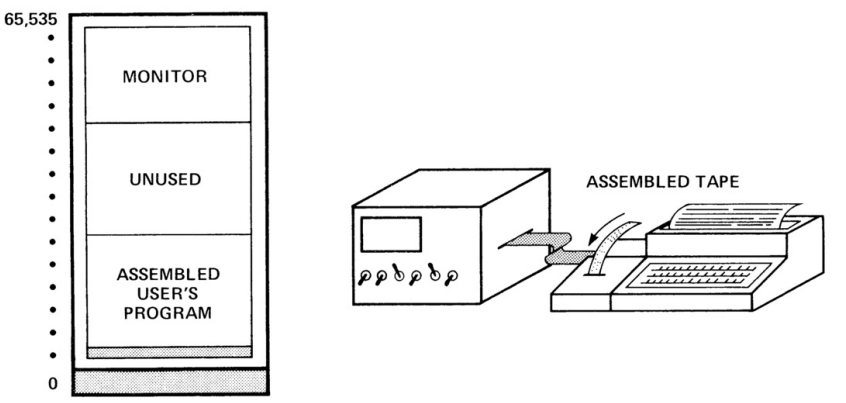


FIG. 4-15 The assembled program tape is read into the system and loaded into the memory so that it can be executed.

Now as the programmer scans the typed page he sees the addresses and machine codes of the program along with any errors and the original program entry for comparison. The only reason for running the second pass of the source tape is to create the hard copy of the assembled program for the programmer's reference. This is referred to as the assembled program listing.

During this second pass of the tape the assembler program also performs any number system conversions so that data expressed in octal, decimal or hexadecimal will be loaded into the memory in binary.

The source tape is now reloaded for the third and final pass through the machine. During this reading of the tape the exact same process is followed as during the second pass. The location counter is incremented and the symbolic table lookups are done in exactly the same way. Now the paper tape punch causes a punch out of the assembled program to be done on another roll of tape. The only things appearing on this tape are the addresses and the machine codes to be inserted into those locations. No errors are listed on the tape nor does the user's original program appear there. The reason for this third pass is to create the tape which will be used to load the system memory with the assembled program. The reason for not doing both the hard copy of the program and the punched tape simultaneously is that most teletype machines cannot type someting independently of what is being punched onto the tape by the punch unit. That is, the punch unit is a slave of the typing unit. If a separate punch unit used with the teletype, both can be done at the same time and the need for three passes of the tape is eliminated.

Once the assembled program has been punched onto tape, it can be loaded into the system memory. This is done by loading the tape into the tape reader and reading it. This is the first time that the user's program appears in system memory in either source or assembled form. As this program is loaded into memory, it replaces the assembler program which is now lost and will have to be reloaded from the assembler tape when it is next needed.

Now, at last, the user's program is contained in the memory. If the system is reset to reset the program counter to 0000H, that program will now execute. This is the moment of truth for the programmer. If the program executed correctly his job is basically done. It only remains to program the PROMs with the

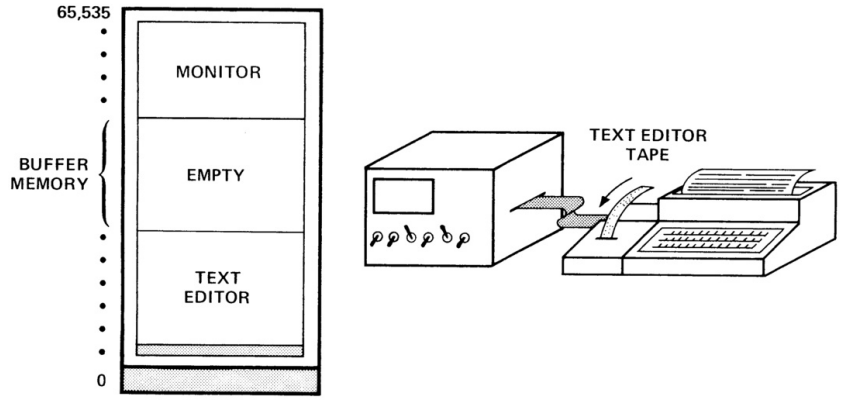


FIG. 4-16 When the errors have been found, the program is ready to be reassembled. To do this it is necessary to use a text editor program which is loaded into the system.

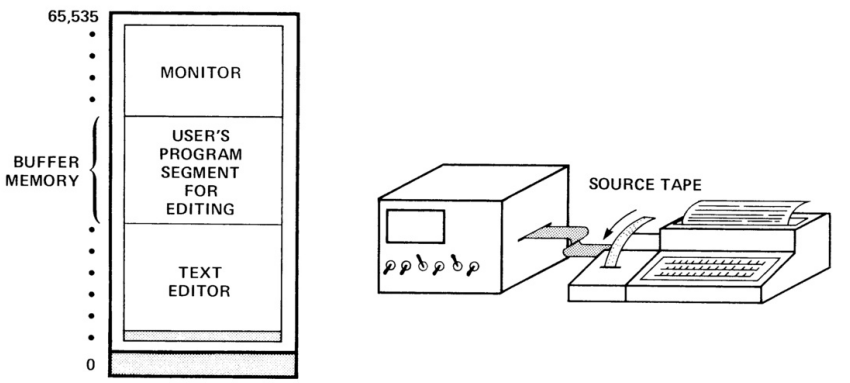


FIG. 4-17 The source tape created earlier is reread, although now only portions can be read at a time. These segments are stored in the buffer memory where the corrections can be made. When each program segment has been corrected, a new source tape of that segment is punched. That new source tape will then be used with the assembler program tape to reassemble the program.

program that was just developed and insert these PROMs into the system. Unfortunately, the chances of having a program run correctly are rare indeed. Then comes the process of debugging.

DEBUGGING THE ASSEMBLED PROGRAM

The debugging process begins with a survey of the source listing, comparing that program to the operation of the system when it tried to execute the program. The hardest program to debug is one that has little physical output; the easiest is the one with the most outputs. This is because more outputs make it easier for the programmer to pinpoint the area of the program cauisng the problem.

Once the problem areas have been located the exact nature of the problem must be determined. Here is where a sophisticated development system is invaluable. An elaborate system, like the lasis ia7700, allows the user to examine each of the working registers and any of the memory locations to determine the problem. Since the programmer knows what the registers should contain, examining them at that point in the program will quickly reveal the problem; alas, the solution is not always so apparent.

Assuming the programmer has located the problem and worked out a solution, he must now somehow get that solution into the program so he can try it. The simplest way to do that is to enter the corrected instructions coded in machine language directly into the memory using the system monitor. To understand how this is done we need to review the status of the system as it now stands.

The assembled program now resides in the RAM portion of the system memory; the assembler itself now exists only on paper tape, having been displaced from the memory by the assembled form of the user's program. The system monitor is still available to the user, since it is permanently contained in PROMs that remain in the system at all times. In addition, the programmer has several copies of the program. The first is the handwritten copy of the source program that he used to enter into the teletype when he typed the source listing. Second, he has the typed copy of the source listing itself. Both of these have one drawback that has probably caused the programmer to discard them; they contain only symbolic addresses in the form of labels. The third program listing however, corrects this deficiency since it contains not only the source listing but the addresses and machine coded instructions as well. This is the program that the user

will examine when he begins the process of debugging. As each trouble area is located he knows by examining the address of the incorrect instruction exactly where in memory the assembler program inserted that instruction.

The monitor contains provisions for inserting new instructions into the memory in place of the instructions already there. To exercise this function the user types the monitor command for Insert followed by the address and the new instruction. The monitor program accepts the ASCII output of the teletype, converts it into binary and reads the binary coded instruction into the specified address. Thus corrections entered into the memory through the monitor must be made in machine coded instructions. If the correct instruction references another address, the binary form of that address must be specified — not the label as was done in assembly language.

You have probably already discovered the problem with this approach to correcting the program. Unless the replacement instructions occupy the same or fewer bytes in memory as the ones they replaced, some of the instructions in the old program will be lost. For example, if six instructions replace three, there will be three instructions of the original program that are lost (assuming all instructions occupy the same number of memory bytes). Of course, replacing six instructions with three is easy; the extra three instructions are made up of NOP (No Operation) instructions that occupy memory locations but cause the system to idle.

The correct approach is to replace the first three bytes of the offending instructions with a JUMP instruction whose jump destination is selected to be an unused memory location at the top of the memory. At that location the full set of correcting instructions are entered, in machine code, and then ended with another JUMP instruction that takes the system back to the original programs at a point that the user believes to be correct as originally written. When the system executed the program, now it will execute as before until it reaches the suspect area, at which point it will JUMP over the offending instructions to a new corrected program segment appended to the end of the original program. After executing the new program segment the system will return to the original program and pick up at that point where the original was believed to be correct. This will take slightly more time to execute, but unless this takes place in a time sensitive loop, which is unlikely, the extra time will not be noticeable. Of course, the program patch also occupies more memory than necessary due to the presence of two 3-byte JUMP instructions. This will be corrected later.

There will probably be many patches required on a typical program. Each time the user patches the program by inserting JUMPs and writing a patch at the end of the original program. As this is done he makes notes of his patches on the assembled program listing to facilitate streamlining the program later. Without these notes he will find it difficult to remove the extra JUMPs when he reassembles the program days later.

THE TEXT EDITOR PROGRAM

At some point the programmer will either reach a point where the program executes correctly or the number of patches exceeds the point at which he can easily keep track of them. At that time he will want to insert the corrected portions of the program into the correct locations and eliminate the extra JUMP instructions. To do this he will make use of a special program called a TEXT EDITOR that is supplied with the development system.

The TEXT EDITOR is a special program supplied on paper tape that is read into the system. Once in memory it replaces the corrected, assembled program that the user has been debugging. This means that the corrections are lost as far as the memory is concerned. They exist however in the notations on the program listing as they were developed by the programmer, and since the original program still exists on paper tape as it was originally assembled, the corrected program can be easily recreated.

The TEXT EDITOR can be thought of as an extension of the system monitor, and indeed, some systems combine both the MONITOR and the EDITOR into one program to save the user the trouble of having to read the EDITOR into the memory. The main difference between the EDITOR and the MONITOR is the EDITOR only recognizes ASCII characters. It performs no operations on the program involving number conversions, calculations, or address assignments. The EDITOR does respond to several commands that may be entered through the teletype keyboard. These have been chosen to be unique and nonconflicting with assembler commands and system commands recognized by the monitor. These include P (Print memory), B (Assign buffer pointer), IN (Insert new character), and so forth.

Once the TEXT EDITOR has been loaded into the memory, the programmer will load the first block of the assembled

program into memory. Since the memory now contains the EDITOR program, plus the fact that the user's program will be stored in memory, comments and all, there will be only room for a portion of the user's program at one time. In typical systems this will be on the order of 50–60 lines of instructions and comments. This program segment is stored in a portion of the system memory allocated by the EDITOR programs and called a buffer memory. The separation of the memory into buffer area and otherwise is made by the software and not by any physical hardware implementations. In fact, the buffer consists of the balance of the system memory not used by the EDITOR and MONITOR.

The contents of the buffer now consist of the exact contents of the source tape, including spaces, carriage returns and line feeds, since they too are ASCII characters. The contents of the tape are stored in the buffer memory in exactly the same order they appeared on the tape, one character per memory location. Since they are punched on the tape in the form of ASCII characters, that is the form they will be stored in the buffer memory. Once the first program segment has been read into the buffer memory, it is ready for editing. This is done by defining a pointer, which is simply a pair of working registers that can be incremented and decremented and perform the same basic function as a program counter. The EDITOR initially established this pointer as location zero in the buffer memory area, that is, the pointer registers contain the address of the first location in the buffer memory. The user then types P (for Print) on the teletype followed by the number of lines he wishes to examine. The EDITOR causes the contents of the buffer to be typed out on th teletype and, since the carriage returns and line feeds are included in the memory, when the system encounters these characters the teletype will perform the desired function. Each carriage return causes a counter to be incremented until the desired number of lines has been typed. The user then examines the program segment to see if he wishes to make corrections in this portion of the program. The listing he examines is, in all respects, a duplicate of the one he typed onto the teletype when he first entered his source program into the system.

If the programmer wishes to make a correction in this portion of his program, he must move the buffer pointer to the offending location. This is most easily done by typing the command to move the pointer to the correct line. If the user then types IN (for Insert) followed by the correcting entry and terminated by an Escape character (a special key on the teletype keyboard), the EDITOR will replace the line with the new line as typed by the user. If the new line is shorter than the original, spaces are inserted in the buffer until the next carriage return character is encountered. The buffer thus contains the original program segment with the exception of the single line that now has been corrected. If the user only wishes to change a character in a line instead of the entire line, he can move the buffer pointer across the line by instructing the EDITOR how many spaces to move the pointer. This allows him to change single characters without having to retype the entire line.

When the program segment residing in the buffer memory has been completely corrected, a new program segment must be loaded into the buffer memory so that it too can be corrected. First, though, the first segment must be saved. This is done by loading a new tape into the tape punch and typing the EDITOR command for punch which causes the contents of the buffer memory to be punched onto the paper tape. When this has been done the punch stops and the system is ready to receive the next program segment for correction. This process is repeated until the entire program has been read into the buffer and corrected. It is then ready for reassembly which is done in exactly the same way as before. The assembler program tape is read into the system memory and the new source tape containing the corrections read in after it. This causes the program to be reassembled with the corrections in place. If the process has been done carefully the program should now execute correctly.

Chapter Five
Hardware Design Casebook

"An 8080-based system on a single PC board in a 3-ring binder."

So far we have discussed the design and operation of microcomputer systems both from an OEM basis and a development system basis. Now we will consider the design of an actual system intended for volume production. In this way we hope to acquaint the reader with the tradeoffs and decisions that go into the design of such a system.

The system under question is the Iasis ia7301 Computer-in-a-Book – an 8080 based system on a single PC board containing both RAM and ROM memory and I/O ports. To ease the user interface the board also contains a 24-pad keyboard and eight 7-segment LED displays, as well as an audio cassette interface. A further design criteria is that the board must measure 8" x 11" so that it can be included in a 3-ring binder along with a text describing the use of the board and material on how to program it.

Programs are to be entered through the keyboard directly in machine language. Hexadecimal keys 0–9 and A–F are provided for this purpose. Eight other keys control the various modes of the system and allow the user to execute programs in either free-running mode or single-step mode, as well as examine and change the various memory locations and registers. A system monitor program is contained in 1024 bytes of PROM; two additional PROM sockets are provided so the user can insert his own PROMs into the system to execute permanent programs of his own. Iasis will provide optional PROMs to fit these sockets that then provide the board with the additional capability of floating point arithmetic, handy programming subroutines and games that can be played on the system.

When the system is first energized it must come up in the monitor mode automatically so that programs can be entered into the memory from the keyboard and executed. The monitor program sets the system up to properly decode the keyboard and perform the desired operation. If the monitor program is not entered automatically when the system is energized or reset, the CPU will not be able to interpret the information from the keyboard and it will be impossible to enter the debug programs.

The user's programs are written into the system RAM memory which has a capacity of 1024 bytes. This allows the user to write programs of considerable length and complexity, probably far in excess of what would ever be used in a training mode which is the intended function of the system. If the extra PROM sockets are used the system capacity is thus 1024 bytes of RAM and 2048 bytes of PROM for a total capacity of 3072 bytes.

The LED displays are driven by the system monitor program to display addresses and contents of memory locations. In another mode they display the contents of the various registers within the CPU such as accumulator, stack pointer, program counter, and the B, C, D, E, H and L registers. The display programs are callable by the user so that he can use the displays for his own programs.

In addition to the keyboard and LED displays, the user is provided with an I/O port dedicated to interfacing to a consumer type cassette tape recorder. The restrictions placed on the system designer were that the system was to work properly with any portable recorder having remote microphone jacks and earphone jacks. No wiring changes to the recorder were permitted. This function is included so that lengthy programs could be placed onto tape and later reloaded into the memory without having to key them in instruction by instruction. This provides the user with the ability to save his memory if he is interrupted and must shut the power off.

In addition, the user is provided with two TTL compatible I/O ports. Although the board was intended to be a programming tool rather than a hardware system, the addition of I/O ports was felt to add to the value of the unit since it allowed the user to interface to outboard peripherals.

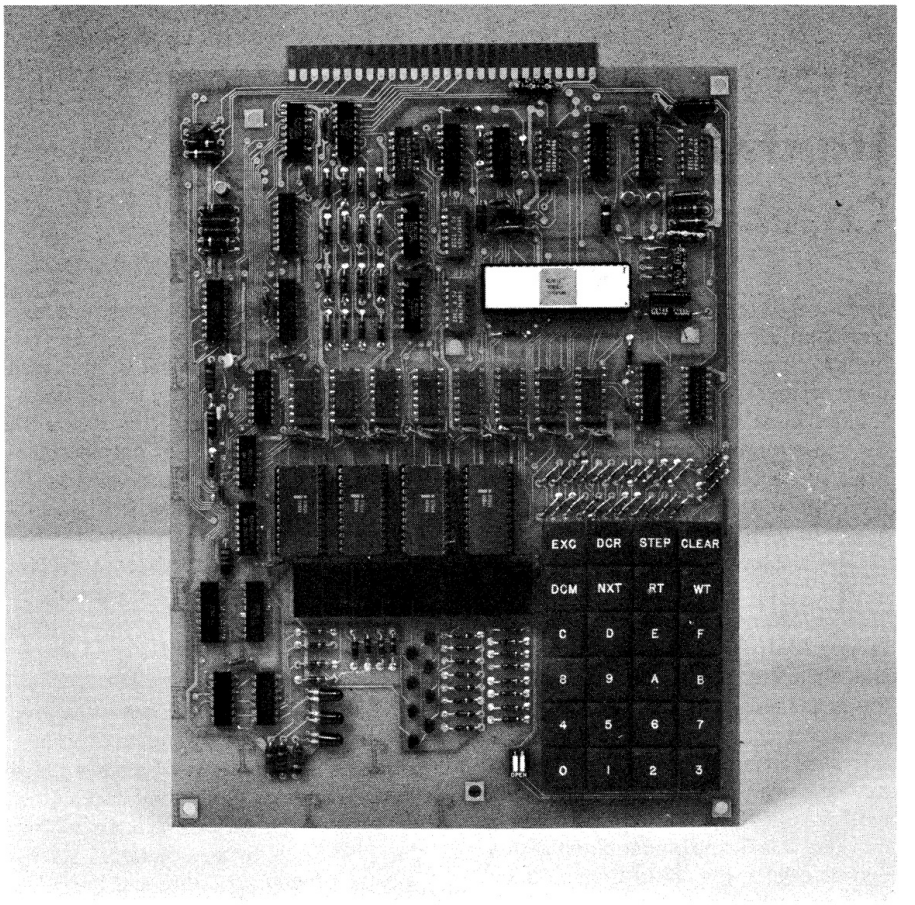


FIG. 5-1 The heart of the Iasis ia7301 Computer-in-a-Book is a complete 8080 system on an 8" X 11" PC board with CPU, clock, RAM, PROM monitor, keyboard and displays.

DEFINING THE MODES OF OPERATION

The natural tendency at this point is to rush headlong into the hardware design by trying to define memory configurations and bus drivers, etc. This is actually a waste of time until the operating modes have been established. Although this sounds like a trivial task, a proper definition of the modes actually settles many of the design questions of the system.

When our system designer reviewed the design criteria he identified seven major modes of operation:

POWER ON/RESET. Whenever the power is first applied to the system or the RESET key is pressed, the Power On/Reset mode is entered. This causes the system to execute the first instructions of the system monitor which initialize the system by setting the Stack Pointer and the program counter both to location 100H (Hex) and all other registers to zero. This mode is transient in the sense that the system exits automatically and enters the DCM mode.

DISPLAY AND CHANGE MEMORY – DCM. In this mode the LEDs display the address and contents of one of the memory locations. The DCM mode is entered automatically from the Power On/Reset mode, at the completion of either a Read Tape or a Write Tape operation or when the DCM key is pressed while the system is in either the DCR or SSP modes.

Entry into the DCM mode results in a display of the DCM mode symbol in the left-most digit followed by the 4-digit memory address and the 2 digits representing the contents of the location addressed. When the mode is first entered the least significant 2 digits of the display, which normally indicate the location contents, will be blank. At this point the user will insert the address of the location he wishes to examine and perhaps change. This is done by simply depressing the appropriate hex keys, 0–9 and A–F (the first key depressed will become the most significant digit of the address). As each numeric hex key is depressed the display shifts the numbers already displayed one digit to the left. If an error is made in entering the address, the user simply keys in the correct address; the incorrect digits will be shifted out of the display and replaced by the correct address.

When the correct address is displayed on the LEDs the operator depresses the NEXT key. The system monitor causes the location addressed by the number on the address portion of the display to be accessed and the contents displayed on the LEDs in the least two significant digits of the display. If the NEXT key is depressed again the address shown on the display will be incremented by one and the contents of the new location displayed. Each time NEXT is depressed, a new location will appear on the display.

If the operator desires to change the contents of the location displayed it is only necessary to key in the appropriate 2-digit number before depressing the NEXT key to bring up the next address. As before, the first key depressed will become the most significant digit. If it is necessary to correct the entry just keyed, the operator keys in the correct digits which in turn shift the incorrect number off the display.

When the address or contents have been entered incorrectly, the erroneous digits disappear as they are shifted out of the appropriate field in the display. Thus, erroneous digits in the contents field disappear as they are shifted out of the contents field; they are not shifted into the field of the addresses where they

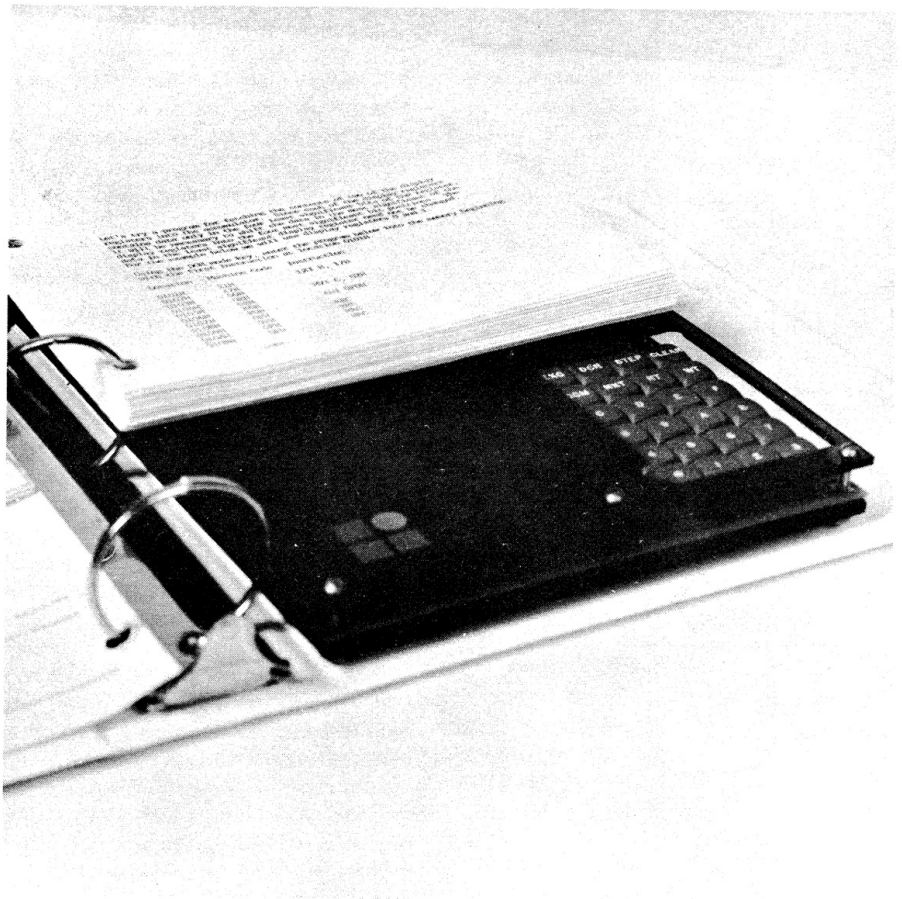


FIG. 5-2 The board is inserted in a 3-ring binder with a complete text on programming the 8080. The size of the pages and the arrangement of the keyboard and displays allow the user to run his programs as he studies the text.

would cause confusion and destroy the address already displayed there.

Once the first address has been keyed into the display, the locations to be examined appear sequentiallly each time the NEXT key is depressed. It is frequently necessary to examine program segments that are many locations apart. To have to hit the NEXT key 500 times in order to jump to a location 500 addresses away would obviously be very inconvenient. This can be done easily by pushing the DCM key again, even though the system is already in the DCM mode. When this is done the address display is blanked and the operator can enter the new address as before.

The DCM mode allows the operator to quickly examine and change any location in memory. It is this mode that is used for writing programs on the system. The DCM mode is entered, the first location, of the new program keyed into the address field, and the first instruction entered by depressing the two hex keys corresponding to the machine code of the desired instruction. The NEXT key is then depressed which brings up the next address, the appropriate numeric keys depressed to key in the second instruction, NEXT depressed, etc. In this manner an entire program can be loaded into the system memory.

DISPLAY AND CHANGE REGISTERS — DCR. During the process of debugging a program that has already been keyed into the system via the DCM mode, it is frequently necessary to examine the various registers on the CPU — the Accumulator, the B, C, D, E, H and L registers, and the Program Counter and Stack Pointer. This can be accomplished by using the DCR mode. As in the case of the DCM mode, the DCR mode is entered by depressing the DCR key while in either the DCM, the DCR, or SSP modes.

When the DCR mode is entered the LEDs display a mode indicator symbol in the left-most digit, the letter A (for accumulator) in the next digit, four blank digits, and two hex characters which represent the contents of the accumulator. New data can now be loaded into the accumulator directly from the keyboard. Whether or not new data have been entered into the accumulator, depressing the NEXT key will cause a new register to be displayed, in this case the B register. Again the contents of the register may be changed by keying in new data before the NEXT key is depressed. In this manner the contents of all of the registers may be examined and changed. The order the registers will appear in is A, B, C, D, E, H, L, SP (Stack Pointer), and PC (Program Counter). If the NEXT key is depressed while the program counter is being displayed, the contents of the accumulator will again be displayed. If the user does not wish to roll through all of the registers to get back to one located lower on the stack, he may be able to shorten the process by reentering the DCR mode via the DCR key. Thus to go from the B register back to the accumulator, it is faster to simply hit tht DCR key a second time than to roll through the intervening seven registers.

This mode must be used to execute a program after it has been loaded into the system memory via the DCM mode. This is because the program counter must be initialized to the address of the first instruction of the user's new program. As already explained, the program counter can be initialized by depressing DCR and then pushing the NEXT key until PC is displayed on the LEDs. The correct address is then keyed into the system.

EXECUTE PROGRAM — EP. This mode may be entered from the DCM, DCR or SSP modes. When the EP button

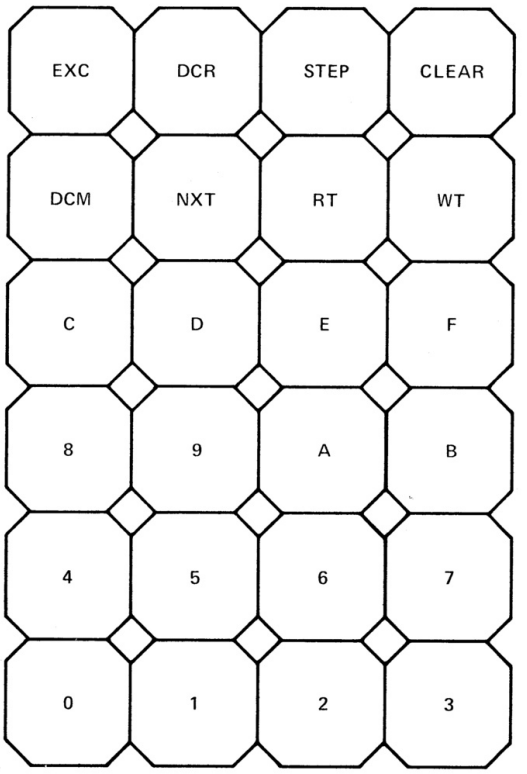


FIG. 5-3 A 24-pad keyboard controls all of the functions of the system.

is pressed, the execution of the program is initiated at the address contained in the program counter. That is why it is necessary to preload this register from the DCR mode. The EP mode is exited in three ways. When either the SSP button is pressed or the RST 1 instruction is executed, the mode is exited by entering the SSP mode. If the RESET button is depressed, a jump occurs to the first instruction of the system monitor which initializes the stack and then exits to the DCM mode. The EP mode produces no display.

SINGLE-STEP PROGRAM – SSP. This mode may be entered from either the DCM, DCR, EP or SSP modes by depressing the STEP key. The next instruction of the user's program will be executed and the system stopped. Ths displays will show the SSP mode indicator symbol, the address of the next instruction and the contents of that location. If STEP is depressed again, the next instruction in the user's program will be executed and the system stopped again. In this manner the program may be executed one instruction at a time. At any point, the DCM or DCR modes may be entered to examine or change either the memory locations or the registers. By switching back and forth between SSP and DCR, the bugs in the program can be found fairly quickly.

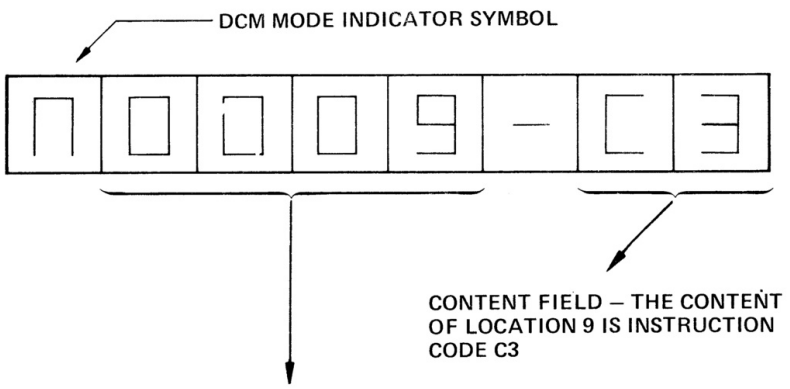
If the SSP mode is entered from the EP mode, the execution of the program will be terminated. If, during the execution of the program a DI instruction, Disabled Interrupts, was executed the STEP key will have no effect since it operates by interrupting the user's program.

WRITE TAPE – WT. After the operator has entered and debugged his program, he may wish to save it. Of course he can write it down and simply key it in every time he wishes to use it. This is undoubtedly what he will do for short programs, but for the longer program of 50 or more instructions this can be tedious and is prone to errors. The system has a output dedicated to writing tapes.

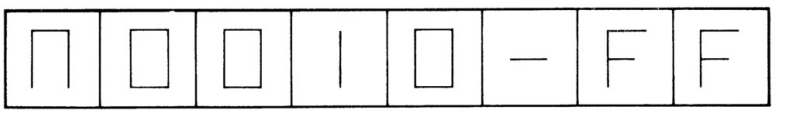
When the operator wishes to use the WT mode to save a program he must first connect the tape recorder earphone plugs and remote microphone plug to the appropriate terminals on the edge connector used to interface to the microcomputer system. All power connections, tape recorder connections and I/O port connections are made via this connector. He places a fully rewound cassette tape into the recorder, sets the recorder in the record mode and starts the tape into motion. He then depresses the WT key on the microcomputer board.

When this mode is entered, the system monitor causes the contents of the memory to be loaded onto the tape. While it is possible to program the system to write and read individual files onto the tape, the monitor as supplied causes the entire 1024 bytes of RAM memory to be written onto the tape. It was felt that manipulation of files was practical only on recorders that could be set into fast forward, reverse, record and play under the control of the microcomputer via solenoid controls. Since the design criteria of the system included the stipulation that it be capable of being used with inexpensive consumer tape recorders, the file manipulation features were omitted.

When the mode is initially entered the system writes a 30-second leader made up of binary ones. At the conclusion of the leader a zero is written, followed by the first byte of the memory, location zero. Naturally, all of the data is in serial form. At the end of each byte is a stop bit which is always a one. At the beginning of each byte is a start bit which always consists of a zero. Hence the data bytes are always separated by two bits, a one followed by a zero. The contents of the memory are loaded onto the tape at the rate of 150 bits/sec. At this rate the entire memory will be loaded onto tape


DCM MODE INDICATOR SYMBOL
CONTENT FIELD – THE CONTENT OF LOCATION 9 IS INSTRUCTION CODE C3
ADDRESS FIELD – LOCATION 9 IS PRESENTLY BEING ADDRESSED

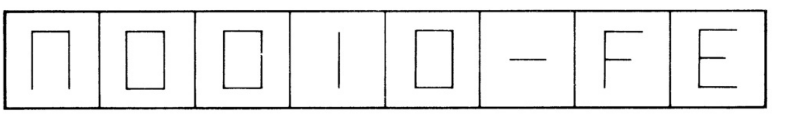
WHEN "NEXT" KEY IS PRESSED, THE NEXT MEMORY LOCATION IS DISPLAYED.



THAT LOCATION, #10, CONTAINS THE INSTRUCTION CODE FF.

TO CHANGE THE CONTENTS FF TO EF, KEY IN THE CORRECT NUMBER, MOST SIGNIFICANT DIGIT FIRST.

AFTER FIRST KEY ENTRY (E):



AFTER SECOND KEY ENTRY (F):

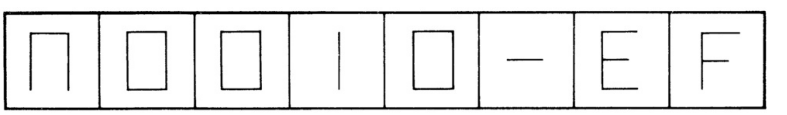


FIG. 5-4 Display during DCM mode.

in about 70 seconds. With the 30-second leader at the beginning of the tape and another 30-second tail at the end of the tape, the entire taping operation requires slightly over 2 minutes.

While there is nothing to separate programs on the tape the user should, with a little care, be able to put several programs on each side of a 30-minute tape. However, there is no way the second program can be searched out on the tape without reading the tape at the normal speed, which slows the entire operation down considerably. With only one program at the beginning of each side the taping operation is still very fast and the low cost of 30-minute tapes seems to make this the most efficient use of tape and time.

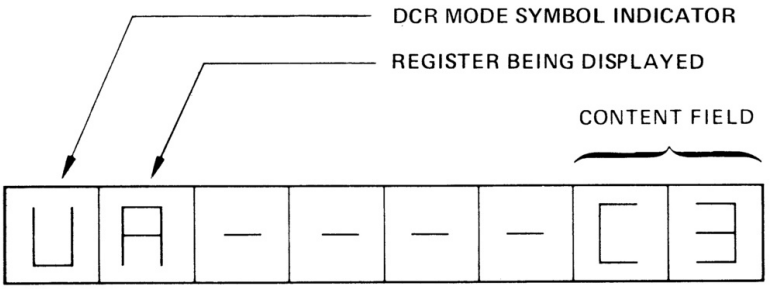
READ TAPE – RT. Obviously it does no good to tape a program if it cannot later be retrieved. That function is accomplished by loading the fully rewound tape into the recorder and setting the tape into motion while in the PLAY-BACK mode. The RT key is immediately depressed and the program will be loaded into the memory. At the conclusion of either a WT or RT operationthe DCM mode is automatically entered.

During both WT and RT no display is produced. This is inconvenient because the user cannot tell what stage the operation is in, that is whether he is reading or writing leader or even if the data is completely loaded. This situation has been provided for by a set of three discrete LEDs which serve to monitor the operation of the system during those modes that produce no display. The EXECUTE LED is lighted when the system is in the EP mode. The WRITE TAPE and READ TAPE LEDs are lighted when in those modes.

MEMORY ORGANIZATIONS

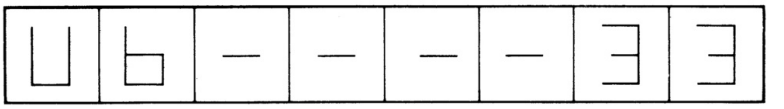
Most microcomputer systems currently being built use memories organized around 8-bit words. This means that there are eight data input lines to the memory and eight data output lines from the memory. Each memory location stores data in the form of 8-bit words, so that each memory address presented to the address lines of the memory will produce an 8-bit data word on the output lines during a memory read. Similarly, a memory write will cause an 8-bit word to be written into the location specified by the address inputs.

There are three common ways of constructing a memory organized around 8-bit words. Memory components may be chosen that store 8-bit words within the

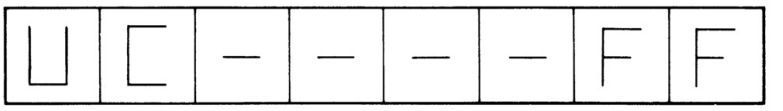


WHEN THE DCR MODE IS FIRST ENTERED, THE CONTENTS OF THE ACCUMULATOR (A) ARE DISPLAYED AND MAY BE CHANGED.

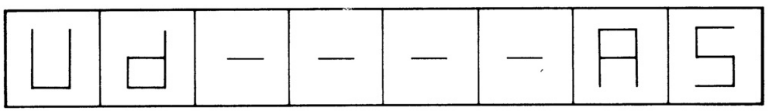
AFTER DEPRESSING NEXT, REGISTER B IS DISPLAYED.



DEPRESSING NEXT AGAIN WILL DISPLAY C:



DEPRESSING NEXT AGAIN WILL DISPLAY D. AT ANY TIME THE CONTENTS OF THE REGISTER MAY BE CHANGED BY ENTERING INTO A NEW NUMBER OF THE KEYBOARD BEFORE DEPRESSING NEXT.



DEPRESSING NEXT AGAIN WILL DISPLAY E:

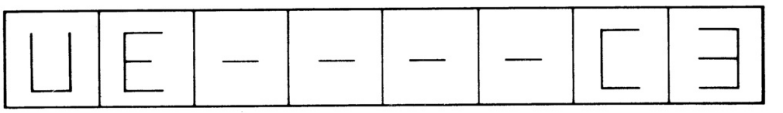


FIG. 5-5 Display during DCR mode.

locations in the component. This is a common organization of PROM, EROM or ROM type memory components. A common PROM is constructed as a 512 x 8 memory component. This means there are 512 locations each storing an 8-bit word. There will be eight data output lines and nine address lines. There are no data input lines since data is not read into a PROM.

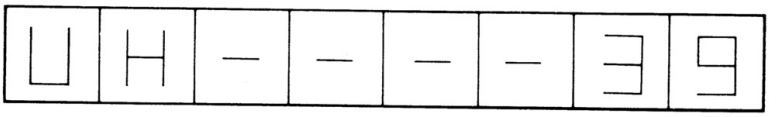
Components organized internally around 8-bit words are very convenient for small memories. Thus a memory built around a component of this type could be constructed with only one device. The PROM just mentioned makes up a 512-byte memory — no other components are required. These components are also very convenient to use when the maximum amount of flexibility is desired. Memory can be added to the system just described in 512-byte increments by adding single components.

An organization now becoming popular in RAMs is the 256 x 4 components. Each of these devices stores 256 words of 4 bits each. To use them in an 8-bit memory system requires two devices for each 256 bytes. Thus one device stores the four most significant bits and the second stores the four least significant bits. Each of these RAMs has four data inputs and four data outputs plus eight address lines.

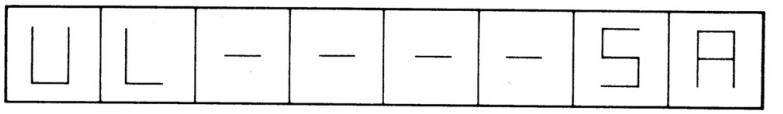
The most common organization for RAM components is the single-bit storage location, either in 1024 x 1 or 4096 x 1 RAMs. The 1024 x 1 devices are static types, making them very easy to use. The 4096 x 1 types have, until recently, been the harder-to-use dynamic type, although recently, static 4096 x 1 types have been introduced. In any event, the user must make a decision as to whether he wishes to use the static types which require little overhead circuitry or the dynamic types which need special clock circuits, drivers and level shifters. In large systems of 16K bytes or more the lower component cost of the 4096-bit dynamic types more than offsets the cost of the extra circuitry, so that small systems tend to be built around the static RAMs and larger systems around the dynamic RAMs.

Having multiple data inputs and outputs on a RAM component increases the cost, so devices organized with multiple-bit words tend to be used only in small memories or where the maximum amount of versatility is needed. To build a memory around 1024 x 1 RAMs requires the devices to be added in multiples of eight. Each of the RAMs then stores a different bit in the data word. The first RAM stores bit 0, the second stores bit 1, the

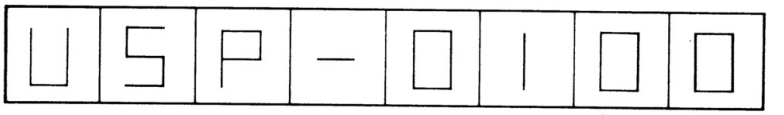
DEPRESSING NEXT AGAIN WILL DISPLAY H:



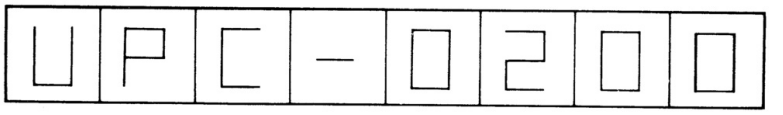
DEPRESSING NEXT AGAIN WILL DISPLAY L:



DEPRESSING NEXT AGAIN WILL DISPLAY THE CONTENTS OF THE STACK POINTER (SP). SINCE THIS IS A 16-BIT REGISTER, FOUR DIGITS OF THE DISPLAY ARE NEEDED:



DEPRESSING NEXT AGAIN WILL DISPLAY THE PROGRAM COUNTER, PC:



PUSHING NEXT AGAIN WILL START THE CYCLE ALL OVER BY DISPLAYING THE ACCUMULATOR:

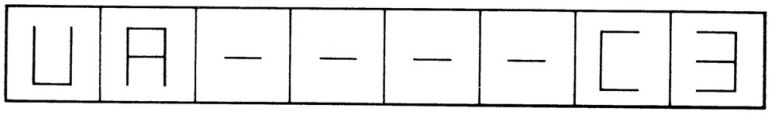


FIG. 5-5 (cont.) Display during DCR mode.

third stores bit 2, and so on. Each has a single data input line and a single data output line. To add an increment of memory to a system using these devices requires the addition of eight more RAMs for an increase of 1024 bytes. To add an increment of memory to a system built around 256 x 4 devices requires only two additional parts which add 256 bytes to the system.

Obviously, systems that require large amounts of RAM memory will use the 1024 x 1 devices to take advantage of the lower component cost. Typical multi-card development systems contain RAM memory cards with 4096 bytes per card. Adding RAM capability to these system is as easy as adding another card. This is a natural application for the 1024 x 1 RAM. Single-card systems, on the other hand, are generally designed for OEM applications which means the system program will be contained in ROMs or PROMs and the amount of RAM storage will be limited to the small amounts needed for scratch-pad storage. For these systems the 256 x 4 is ideal; the card may be supplied with eight sockets. The user plugs in two, four, six or eight RAMs for a capacity of 256, 512, 768 or 1024 bytes — a very flexible arrangement.

DEFINING THE MEMORY AND I/O STRUCTURE

With the seven basic modes of operation defined, the next step is to consider the structure of the memory and I/O ports. With large systems, this is very straightforward. A large portion of the memory addressing capability of the CPU will be used and the only question is whether to treat the I/O as memory locations or to use the direct I/O lines of the 8080 and access the ports through dedicated Input/Output instructions.

The proposed system is actually fairly small and it is made simpler by the fact that no DMA activity is required. Systems this small offer another possibility — that of using unused memory address lines to control the modes of operation.

Such is the case in this system. Of the 16 address lines that come from the 8080, only 10 of them are used to directly address the RAM memory. The remaining 6 are used to control other aspects of the system.

Since the ia 7301 was to be supplied with 1024 bytes of RAM in a nonexpandable memory configuration, the use of the more economical 1024 x 1 RAMs was the natural choice.

This leaves the question of how to access the I/O ports. Obviously they

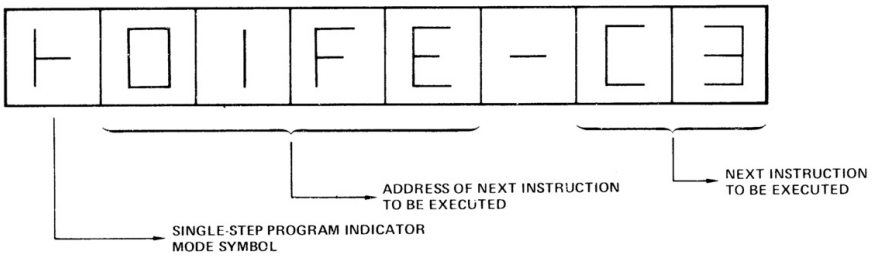

H 01FE-C3
ADDRESS OF NEXT INSTRUCTION TO BE EXECUTED
NEXT INSTRUCTION TO BE EXECUTED
SINGLE-STEP PROGRAM INDICATOR MODE SYMBOL

FIG. 5-6 Display during Single-Step Program mode (SSP). Each time STEP key is depressed the instruction shown will be executed, the address incremented and the next instruction displayed.

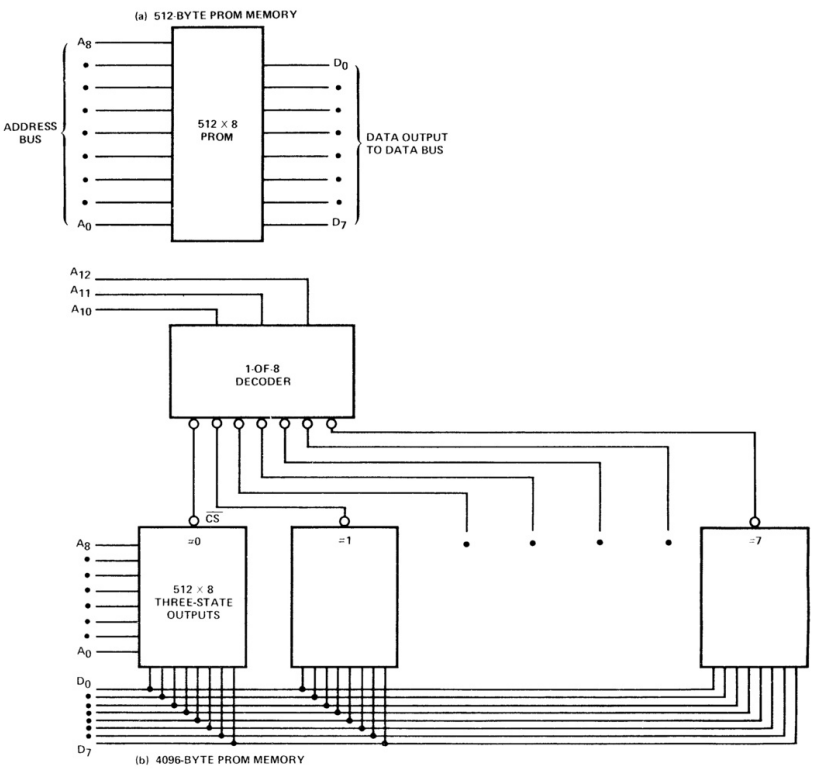


FIG. 5-7 Memory designs using 512 X 8 components.

could be tied to the I/O command lines which would then enable the port being selected by the address on the address bus. This technique requires the programmer to use the Input and Output instructions provided in the 8080 instruction set; when these instructions are executed the I/O command lines are activated. Since separate command lines are used for I/O and memory operations, the user need not concern himself over the chance of having the same address used for both I/O port and memory locations; the 8080 will keep the operations straight by its use of the command lines.

An alternate approach is to treat the I/O ports as memory locations that can send or receive data by executing a MOV or MVI instruction to the proper address. In this case the I/O command lines and Input and Output instructions are not used. Needless to say, the addresses chosen for the I/O ports must be unused by the memory or every I/O operation would also cause data flow within the memory.

Which I/O technique is used depends on the requirements of the system. In very large systems requiring most of the addressing capability of the 8080, treating the I/O ports as memory locations may require decoding circuits for the I/O ports that are actually more complex than if the standard I/O command lines had been used. On the other hand, in small systems, treating the I/O ports as memory locations eliminates the need for logic to implement the I/O command lines. In many instances programs can run faster with memory mapped I/O since the programmer is not restricted to simple Input and Output instructions with which to move data through the I/O ports. Instead, he has a whole set of instructions at his disposal, many of which can perform logical operations on the data as it is moved, and thus save instructions and shorten the execution time.

It is not clear whether the designers of the 8080 saw the possibility of treating I/O as memory locations, since the circuits from Intel all use I/O command lines. Still, all of these circuits will operate just as well with either the I/O command lines or the memory mapped I/O techniques. Probably, the designers wanted to allow for the very large systems by including I/O command lines. Experience now shows that there are very few systems indeed that cannot use memory mapped I/O as an advantage, and the second generation 8-bit microcomputers like the Motorola 6800 and Fairchild F-8 all use memory mapped

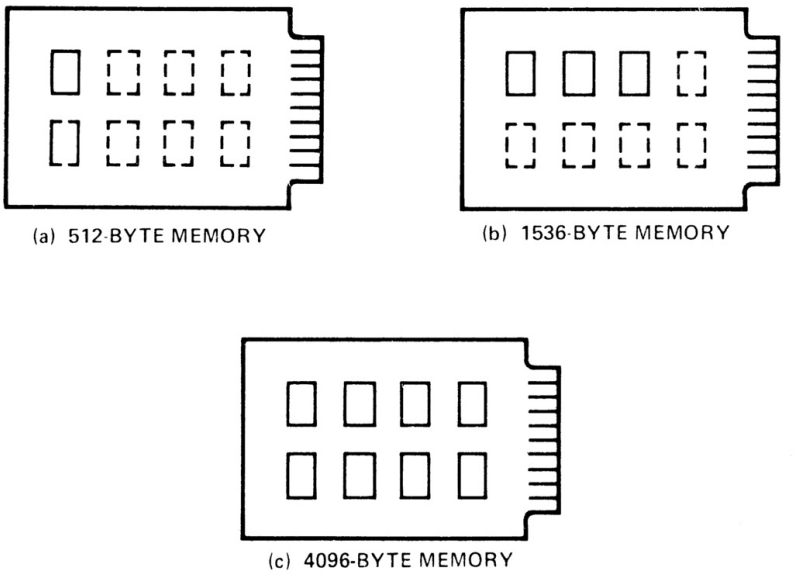


FIG. 5-8 Components organized as 8-bit words can be combined with sockets to make memory systems that the user can expand in small increments.

I/O exclusively.

In the case of the ia7301 system several techniques were combined. The memory address lines were only partially decoded, a technique that saves logic elements at the expense of unneeded memory. This technique was then combined with the use of memory mapped I/O to reduce the amount of logic elements to the minimum. To use this technique effectively the designer must first identify the types of memory and I/O operations that the system must be able to perform. This is very similar to the identification of the operating modes that was done earlier in this chapter. Obviously identifying the configuration of the memory blocks and I/O ports requires some preconceptions of the overall design of the system. Thre is no formal way of attacking this problem, other than a lot of thought and willingness to try all the alternatives.

Our designers grouped the functions in such a way that five distinct memory-I/O blocks emerged:

RAM. Because of the partial decoding of the address lines the 1024 bytes of RAM in the system will appear more than once in the system memory. That is, the first RAM location can be accessed by addressing location 0000H, 2048H, etc. Because the RESTART instructions that are executed when an interrupt occurs must appear in the first 64 locations of the system memory, if the user is to be able to write his own interrupt routines, the low order locations must be RAM. Since the interrupt routine was to be supplied with the ia7301 system, the requirement that the low-order addresses be RAM was not a stringent one. Nevertheless, systems with the RAM at the bottom of memory are so common that our designer bowed to convention and executed the first 1024 bytes of the system as RAM.

PROM. The 2048 locations reserved for the system monitor PROMs may appear anywhere in the memory, but they must somehow be the first instructions fetched when the system is energized so that the system comes up with the monitor in control. There are three ways of accomplishing this. First, the operator can write a JUMP instruction into the first three bytes of system memory using DMA techniques. Then when the RESET is hit, the system executes the JUMP instruction which causes a JUMP to the system monitor. This was the method used in the first Intellec development system. Second, the first locations can be duplicated in both PROM and RAM. When the system is energized the PROM is active and the monitor is begun. At the end of this PROM is a JUMP to the remainder of the monitor which occupies PROMs at the top of the system memory. When the user's program is executed this first PROM is disabled and the RAM in the same location is enabled. This is the technique used in the current series of Intellec MDS development systems. A third technique, and the one used in the ia7301 system being designed here, is to modify the address sent out by the 8080 when the system is first energized. This causes the high-order PROMs to be accessed even though the program counter in the 8080 is sending out location 0000H on the address bus. In small systems this is considerably simpler than the other two approaches.

KEYBOARD-INTERNAL INPUT/DISPLAY-OUTPUT. When this block of memory is addressed, two operations will be performed simultaneously. The keyboard will be accessed and the information contained in the display latches will be displayed on the LED displays. To simplify the operation of the system, the keyboard is set up as a matrix of switches. An output operation must be performed to a row of the keys and then an input operation done to see if any of the keys in that row were closed. Then the next row of keys is activated with an output operation and an input performed to

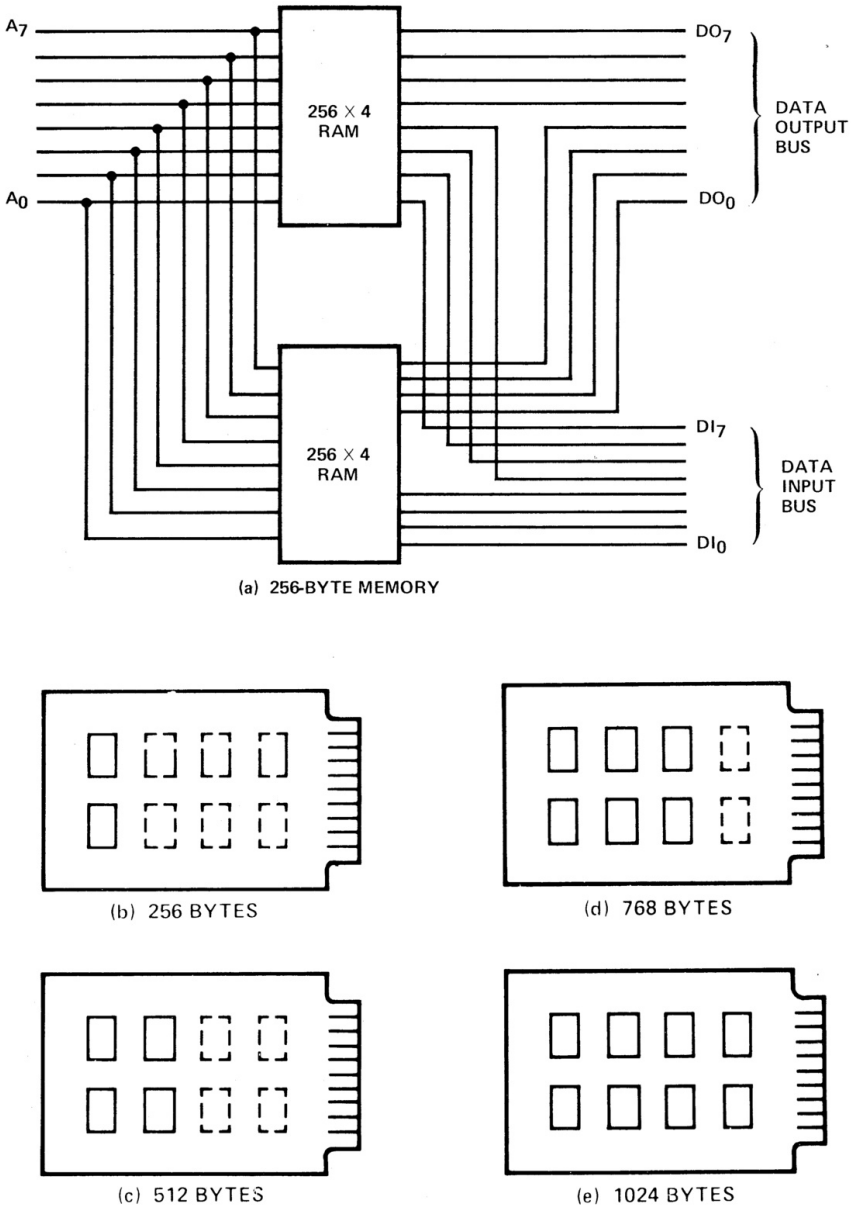
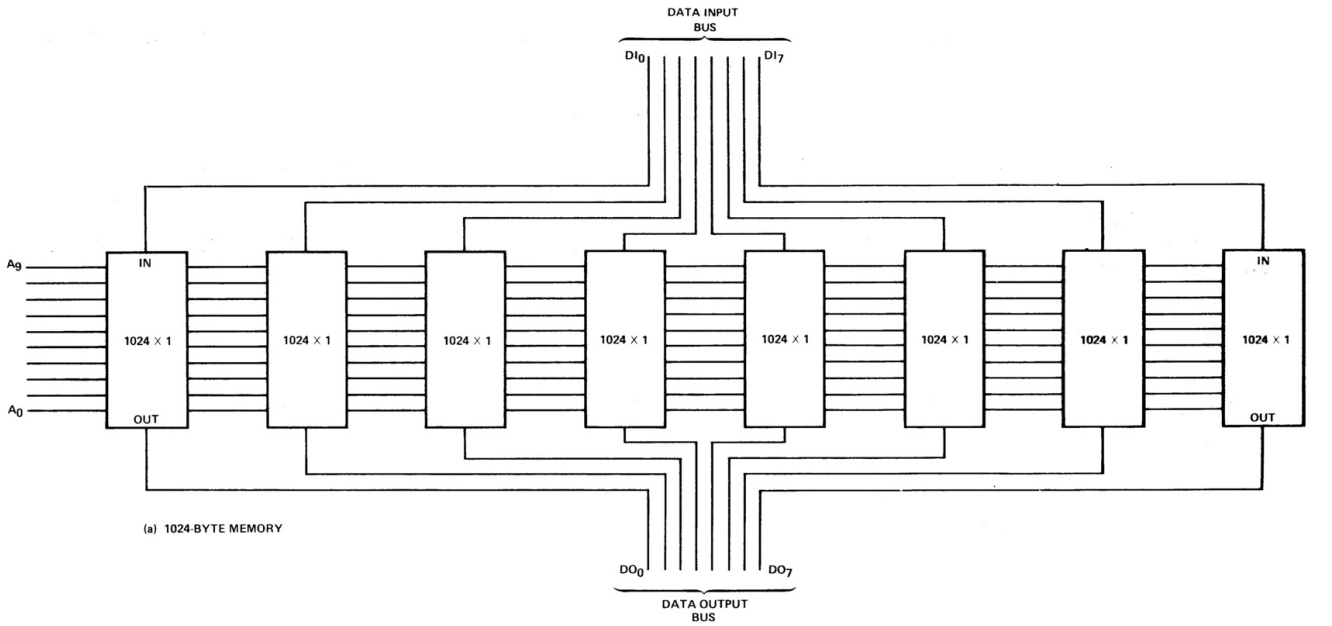


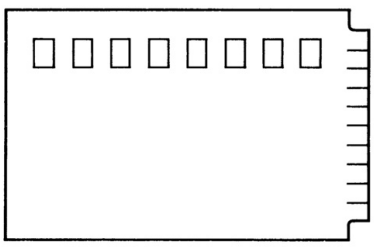
FIG. 5-9 Memory systems built with 256 X 4 RAMs can be expanded in 256-byte increments.

check for key closures in that row. In this way, all of the keys can be checked with only two ports, an Input port and an Output port.

The same scheme is used for the displays. The information to be displayed is latched into an output latch that drives the 7-segment displays. The outputs of this latch are wired in parallel to all of the displays, but only one of the displays may be activated at one time. The activation of the displays is done by the same output port that activates the rows of keys in the keyboard. Thus a row of keys is activated (simultaneously activating one of the display digits which then displays the number contained in the display latch). If no key closure is found, the next row of keys (and the next display digit) is activated. The number in the display latch is replaced with a new number corresponding to the next digit display. This number is then displayed while the presence of a key closure is checked. In this fashion each row of keys is checked and each of the LED digit displays is activated in turn. The program must check the input port corresponding to the keyboard and must load the new number to be displayed into the display latch. While each of the displays is enabled for a short time, the display current is increased by making the current limiting resistor in the driver appropriately small. Hence the brightness is increased; the scanning of the keyboard/displays is done so rapidly there is no flicker and the effect to the user is that all of the displays are lighted. By combining the keyboard and display hardware and using this scanning technique, only three ports must be used; one output port to drive the key rows and enable the display digits, one input port to check for key closure in the activated row, and one output port to load the display latch that will drive the LEDs. Since the port that drives the key rows has eight outputs, the keys will be arranged in an 8 x 3 matrix. That means that each row activated by the output port contains three keys which are then checked for closures by the input port. With only three keys to check, there are five unused positions in the input ports; these are used for other functions which will be explained shortly. The result is that only one input port is needed for all of the system monitor functions — keyboard and tape read inputs.

INTERNAL OUTPUT. There is one operation that requires an output port treated as a separate address. This is the Tape Write operation. Each time this




(b) 1024-BYTE MEMORY

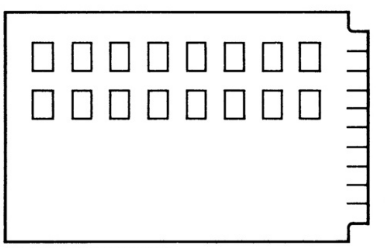

(c) 2048-BYTE MEMORY

FIG. 5-10 While the 1024 X 1 RAM is the most common, the fact that eight devices must be used to make up the 8-bit word restricts the 1024 X 1 RAM to larger memory systems.

port is addressed a signal is written onto the tape (assuming the recorder is in operation and is in the RECORD mode). This port is different than the output port used to load a number into the display latch, since that port was enabled every time the keyboard was driven and did not, therefore, require separate address decoding circuits.

EXTERNAL I/O. The user is provided with one Input port and one Output port. Since the system is designed using memory mapped I/O techniques, the user I/O ports will be accessed every time a certain portion of the memory is addressed. Thus if the address of the user's I/O ports is 5XXXH any address beginning with 5 will access the ports. If the programmer loads the H and L registers with 5000H, for instance, and the A register with the data to be output, executing a MOV M,A will cause the data in the accumulator to appear on the outputs of the output port. If the system then executes a MOV A,M instruction, the data on the inputs of the input port will be loaded into the accumulator, replacing the prior contents.

With these five blocks of memory-I/O in mind, we can assign addresses. This has been done in Figure 5-11. The four MSB address lines are used to select the five memory-I/O blocks. In the case of the Keyboard-Internal Input/Display Output mode, the next three MSB address lines are used to select the keyboard row/digit display to be activated. The rationale for the address selections was to keep the change between RAM and PROM modes down to a single bit change to minimize the hardware necessary to bring

	A_{15}	A_{14}	A_{13}	A_{12}	A_{11}	A_{10}	A_9	A_8	A_7	A_6	A_5	A_4	A_3	A_2	A_1	A_0
RAM	0	0	0	0	0	0	A	A	A	A	A	A	A	A	A	A
PROM	1	0	0	0	0	A	A	A	A	A	A	A	A	A	A	A
KEYBOARD-INTERNAL INPUT/ DISPLAY OUTPUT	1	1	0	0	S_2	S_1	S_0	X	X	X	X	X	X	X	X	X
INTERNAL OUTPUTS	0	1	1	0	X	X	X	X	X	X	X	X	X	X	X	X
EXTERNAL I/O	0	1	0	1	X	X	X	X	X	X	X	X	X	X	X	X

FIG. 5-12 Memory I/O Map for the ia7301. If a WRITE operation is being performed, S_2 S_1 S_0 selects the keyboard row/display digit.

the system up in the monitor mode when the power is applied. Another consideration applying to all of the address lines was to spread the circuit loading equally to all of the lines. This was done in order to eliminate the address bus buffers commonly required in 8080 systems. In the ia7301 the 8080 is able to drive directly all of the memory and I/O port address lines.

RAM MEMORY BLOCK

The address assigned to the RAM memory is 0XXXH. Actually only the two high-order address lines, A_{15} and A_{14}, are monitored to select the RAM section of the system memory. These two lines must be inverted and then applied to the inputs of a two-input NAND gate which is then tied to the CS inputs of all eight of the 1024-bit RAMs. It turns out that the inverted outputs of the A_{15} and A_{14} lines are needed by some of the other portions of the system so the inversions required by the RAM are convenient. We can think of the address bus as now being made up of 18 lines, A_0 through A_{13}, A_{14}, A_{15} and A_{15}. In addition to the gate required by the RAM CS inputs, a WRITE signal must be applied to the RAMs when data is to be written into them. Although a WRITE signal of the correct logic sense for the RAMs is provided by the 8080, some of the other portions of the system require the opposite logic sense, WRITE, and these would load the 8080 output excessively. Thus the WRITE output of the 8080 is buffered twice to provide a signal with the correct polarity and sufficient drive for the RAM section of the system.

Each of the RAMs is used to store a different bit of the 8-bit byte. Each has a separate data input pin and a data output pin. The inputs, when grouped together, form the input bus to the memory. Likewise, the data outputs, when grouped together, form the output bus from the memory. In order to eliminate confusion, the buses are labeled relative to the 8080 and not to the memory. Thus the 8080 outputs data to the memory via the OUTPUT bus which is tied to the RAM data inputs. When the 8080 reads data from the memory it does so on the INPUT bus which is tied to the RAM data outputs. This convention is used throughout this book.

When the 8080 is interrupted the system must force a new instruction onto the INPUT bus. In the ia7301 an interrupt is used for one purpose only – forcing the execution of a single instruction during the SINGLE STEP PROGRAM mode. No other interrupts are allowed, and there are no interrupt inputs brought off the board for the user. In order to minimize the logic elements in the system, the instruction our designer chose to be jammed onto the bus during an Interrupt Acknowledge cycle was RESTART 7 which happens to be all 1's. Thus to jam the RESTART 7 instruction onto the bus all that is necessary to to float the INPUT bus. To accomplish this, eight two-input NAND gates are used. When their common input is pulled low the bus is floated and the desired RESTART 7 is forced onto the bus.

The NAND gates that force the RESTART 7 complicate our life in one regard. They cause the data from the memory and the input ports to be inverted before it reaches the 8080. To compensate for this inversion on the INPUT bus it is necessary to add inverters

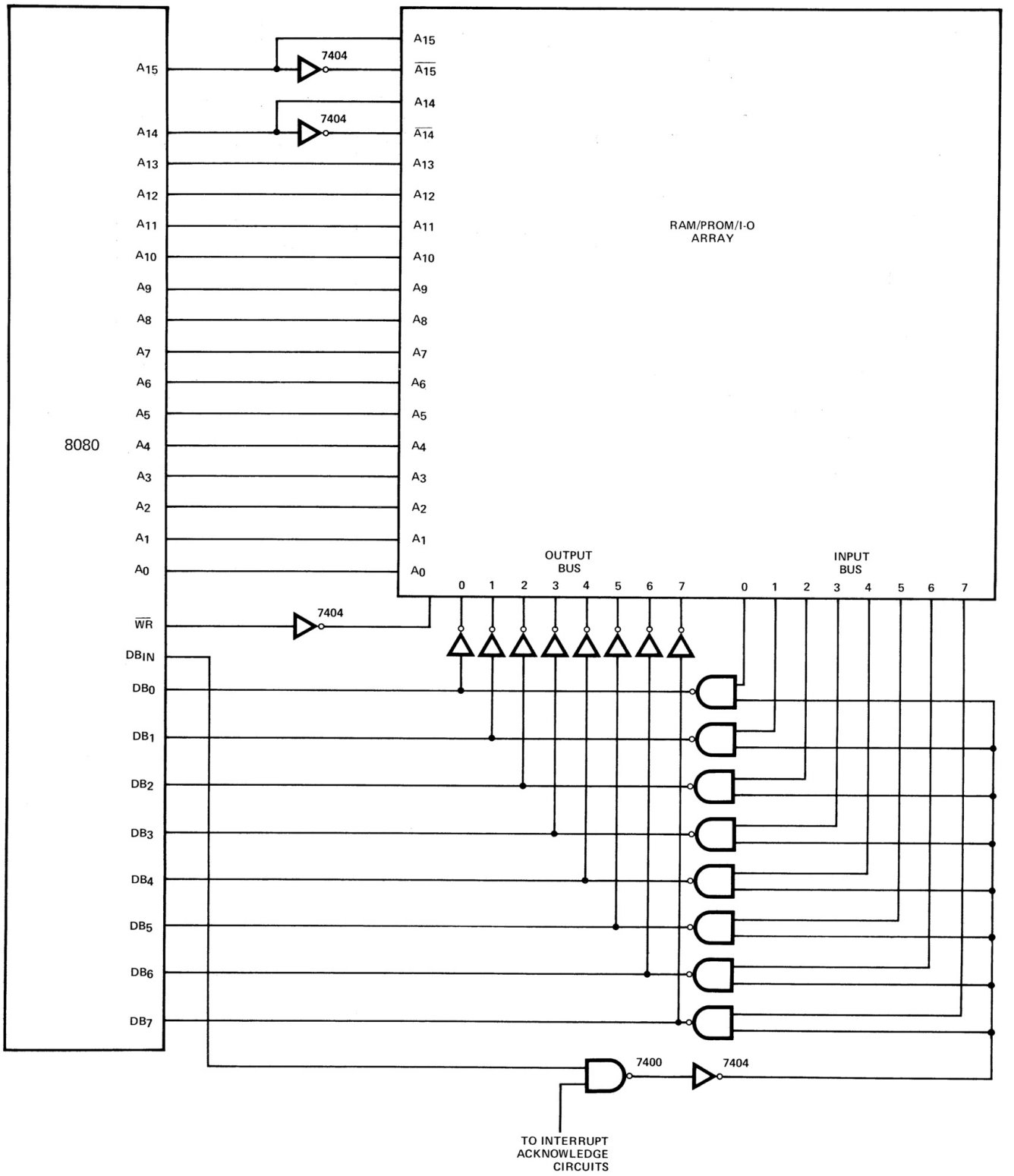


FIG. 5-13 Address and data bus buffers for the Address, Input and Output buses.

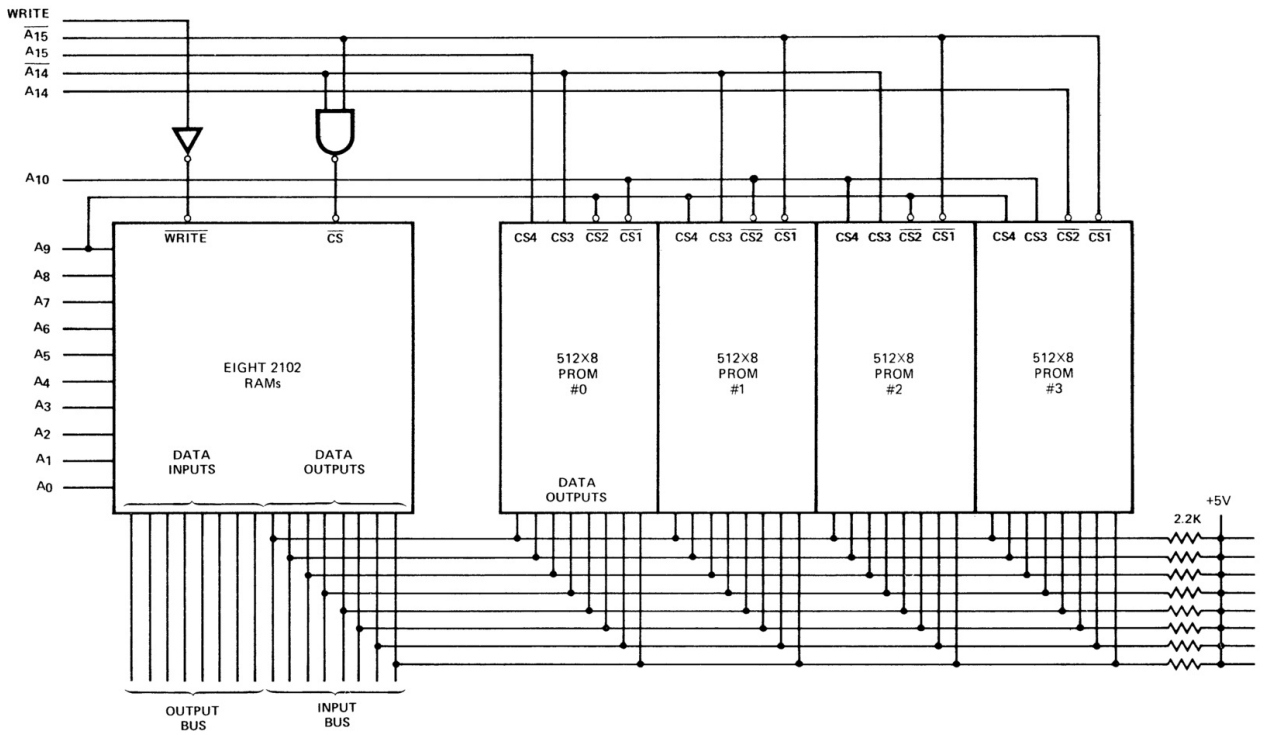


FIG. 5-14 Memory system of the ia7301.

on the OUTPUT bus. These also serve to buffer the data bus outputs of the 8080 eliminating any drive problems with the TTL circuits in the I/O blocks. The drive circuits for the INPUT and OUTPUT buses are shown in Fig. 5-13.

PROM MEMORY BLOCK

There are two separate addressing problems that must be handled by the chip select inputs of the PROMs. One is to enable the PROMs only when A_{15} and $\overline{A_{14}}$ are selected. The other is to select the one PROM out of four as specified by the address lines. Fortunately the PROMs used in the ia7301 have four chip select inputs, two active low and two active high, so the PROMs can be connected to the address bus without requiring extra gates or inverters.

Because each of the PROMs must have a unique address a different combination of chip select inputs was used for each. This complicated the problem of having the PROM block selected by A_{15} and $\overline{A_{14}}$ since in some cases the necessary chip select inputs had already been used. However the fact that connecting $\overline{A_{15}}$ to a $\overline{CS}$ is the same as connecting A_{15} to a CS allowed all of the chip select inputs to be used in such a way that no extra gates were required. The entire memory block of both RAM and PROM is illustrated in Fig. 5-14.

POWER UP-RESET

As has been pointed out several times the system must come up with the monitor program in control of the system when power is applied. This requires the addition of some circuitry in the A_{15} address line as supplied by the 8080. When power is first applied the reset circuit applies a reset pulse to the 8080 and the I/O registers. When this pulse is removed the program counter sends out its contents, 0000H, as the first address on the address bus. 0000H is, of course, the first location in RAM.

The reset pulse from the power up circuit is used to set a flip-flop that changes the A_{15} address line from a 0 to a 1. This means that the memory sees 8000H as the address being supplied and it fetches the contents of that location which happens to be the first location in PROM and hence, in the system monitor. The flip-flop continues to modify the A_{15} line for the first three memory fetches. These three bytes contain a JUMP to location 8003H, which is the next location in PROM. As soon as the

third byte is received the 8080 executes the jump. This results in the program counter sending out 8003H on the address bus; A_{15} thus carries a 1. When A_{15} goes high in response to the JUMP instruction, the flip-flop is reset and the A_{15} address line is no longer modified. Now the memory sees the exact address as sent out by the program counter. The flip-flop is now out of the picture and remains so until the CLEAR key is depressed which starts the cycle over again.

The power up-reset circuit is shown in Fig. 5-15. This circuit uses two inverters in a positive feed-back mode to sharpen the voltage transition through the logic threshold. When power is first applied R_1, R_2 and C form a voltage divider. The sharp rise in V_{cc} results in most of the voltage appearing across R_1 and R_2. The low voltage drop across C causes the input to inverter A being low. A inverts this signal causing a high output which is applied to the $\overline{\text{RESET}}$ input of the 8080. This signal is further buffered to provide a $\overline{\text{RESET}}$ pulse which is used in the address modification circuit on A_{15} and to clear the latches in the system I/O ports. As the capacitor charges up it redevides a larger share of the voltage applied to the divider until it finally reaches the threshold of inverter A. When this happens A inverts its high input resulting in a low on the output. This removes the $\overline{\text{RESET}}$ signal from the 8080 which begins to send out its first address, 0000H, and the $\overline{\text{RESET}}$ signal so the I/O ports can now accept data.

There is a potential problem with this little circuit that forces the addition of inverter B. As the voltage across the capacitor rises and approaches the threshold of inverter A, the noise immunity at that point is going to zero. If, after crossing the threshold and causing inverter A to switch, a negative-going noise spike of only a few millivolts occurs, the voltage at the input of the inverter will drop back through the threshold for the duration of the spike. This causes a second RESET pulse out of inverter A which could potentially cause a problem. Adding inverter B removes this possibility. When power is first applied to the system the input to A is low, the output of A is high which forces the output of B low. At this time the current through R_1 is actually made up of two currents, one through the R_2-C combination and the other through R_3 and the output of inverter B which represents a path to ground when the output is low. When the rising voltage across the capacitor forces A to switch low, inverter B must switch high which removes the second current path through R_3. This causes a fast, sharp reduction in the current through R_1 which in turn reduces the voltage drop across it. Reducing the voltage across R_1 has the effect of increasing it across C; the input to inverter A sees a sudden increase past its threshold which eliminates the effect of noise on the system. See Figure 5-16. The CLEAR key has the same effect on the system as did applying the system power. When the key is depressed it shorts the charge on the capacitor to ground. When the key is released the cycle procceeds in the same fashion as before.

The entire power on-reset circuit is shown in Figure 5-17. Inverters A and B make up the power on circuit as before and supply the RESET pulse to the 8080 when either the CLEAR key is depressed or power is first applied. The output of inverter A is buffered by C which supplies a $\overline{\text{CLEAR}}$ pulse to the rest of the system which requires an active-low signal. The output of C is also used to set the flip-flop made up of gates F and G which are wired in the familiar cross-coupled latch configuration. During the RESET pulse the output of G is latched low, and this signal is applied to the input of gate H forcing its output high. It is the output of gate H that is applied to the A_{15} address line of the memory. Thus when the system is RESET, A_{15} will be high which causes the system to come up in the monitor. Inverters D and E invert the outputs of the 8080. In particular the D inverter changes A_{15} from 0 as sent out by the 8080 to a 1 which cannot force H low. The overriding function then is the output of G which is 0 and forces H high.

When the 8080 executes the JUMP to 8003H that is the first three bytes of the monitor, A_{15} goes high. D inverts this to apply a 0 to the input of H. Gate H now has two 0's on its inputs and remains in the high state. The output of D is also tied to one of the inputs of G, and the low output of D has the effect of resetting the flip-flop and removing one of the 0's from H. The flip-flop can only be set again by a CLEAR signal caused by a key depression or a power-up signal. The flip-flop thus remains out of the system after the first three bytes are executed. When A_{15} from the 8080 now goes low as data is retrieved from RAM, the memory will follow it.

CLOCK CIRCUIT

The 8080 requires two clock pulses, non-overlapping, of 12V amplitude. At full speed, the clock frequency must be 2 MHz. There are several ways to accom-

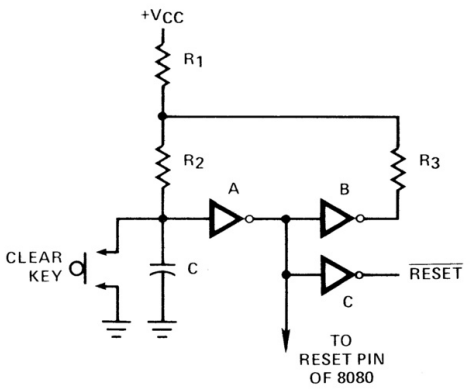


FIG. 5-15 Power-up circuit.

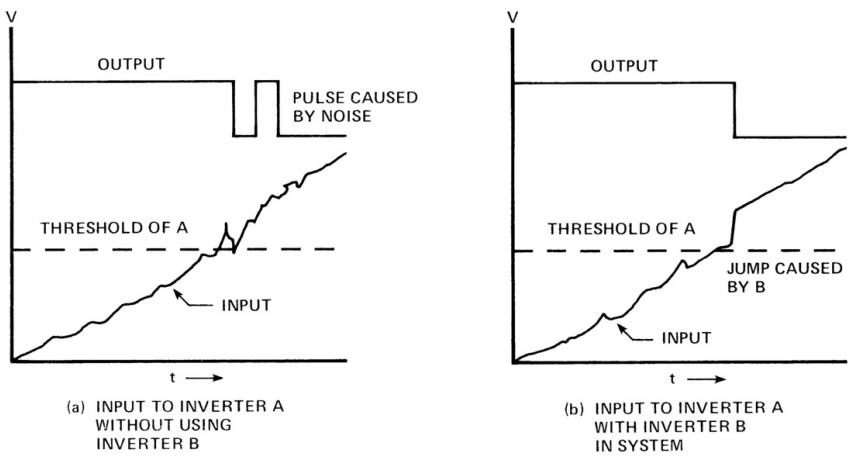


FIG. 5-16 Adding inverter B removes potential noise problems from the power-on/reset circuit.

plish this, ranging from crystal oscillators with special driver circuits to monolithic clock generator ICs made especially for the 8080. The method used with the ia7301 is one of the simplest and cheapest. It uses an 8 MHz crystal in an oscillator circuit made up of two inverters wired as linear amplifiers. See Figure 5-18. The output of the oscillator is buffered by inverter C and applied to the clock inputs of two JK flip-flops, D and E. The J input of flip-flop D is wired high and the $\overline{K}$ input is wired low. In this configuration the D flip-flop will toggle on every clock pulse. The J input of the E flip-flop is wired to Q of D, and the $\overline{K}$ input of E is wired to $\overline{Q}$ of D. In this mode the E flip-flop will toggle when the Q output of D goes low.

The $\overline{Q}$ output of E is buffered to porduce the $\phi 2$ clock. First, however, it must be delayed a few nanoseconds to meet the nonoverlapping requirement. This is done by R_1, C_1, and inverter G. The output of the inverter is then applied to the inputs of three 7406 open collector inverters. The outputs of the inverters are wire-OR'd together to meet the high sinking requirements necessary to provide a fast fall time on the clock. The outputs have a common pull-up resistor tied to 12V to meet the clock amplitude spec.

Because the E flip-flop toggles with about a 50% duty cycle, this is the width of the $\phi 2$ clock pulse. The flip-flops together are wired in a divide-by-four mode, making the total clock frequency 2 MHz or a period of 500 ns. The 50% duty cycle thus establishes $\phi 2$ as about 250 ns.

The generation of $\phi 1$ is slightly more involved, requiring the use of a two-input NAND gate. One of the inputs is tied to the Q output of the D flip-flop. The other is tied to the $\overline{Q}$ output of the E flip-flop. The output of the gate, H, will go low whenever both of its inputs are high, that is, when D is high and E is low. This happens every fourth state so the resulting $\phi 1$ clock runs on a 25% duty cycle or 125 ns. See Figure 5-19.

CHARGE PUMP CIRCUIT

The 8080 is often faulted over other microcomputers because it requires three supply voltages, +5V, +12V and -5V. These criticisms are frequently overstated. The TTL logic in the system requires +5V so that supply must be present anywhat. The current drain from the +12V supply is minimal and can be met with an inexpensive supply. In many small systems the 8080 will be the only device requiring -5V and its drain is in the microampere region. This voltage can

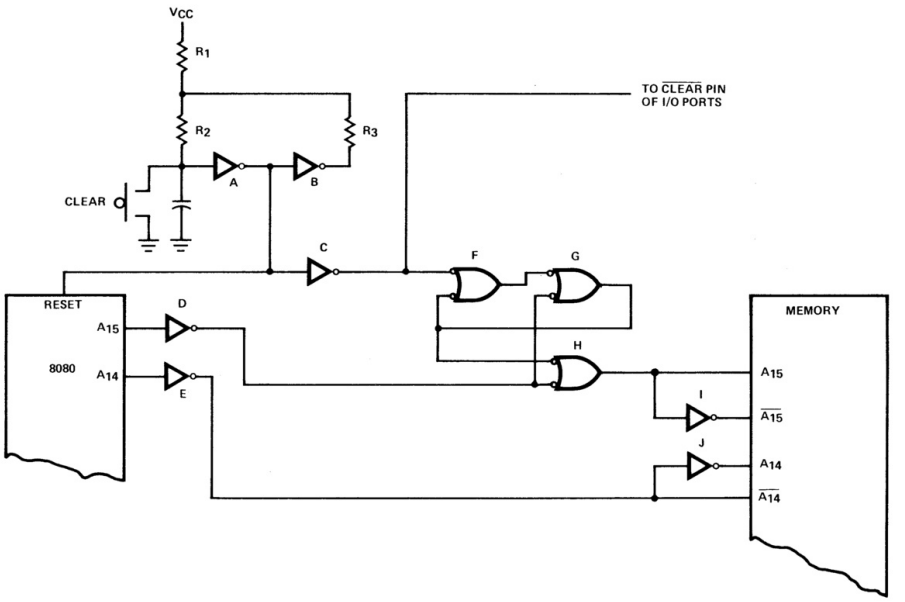


FIG. 5-17 Power-on/reset circuit and flip-flop added to address bus to bring system up in monitor.

easily be generated from the +12V supply with a pair of transistors and a few diodes and capacitors. In larger systems 8K EPROMs are appearing more and more often and these require –5V with moderately large currents, so in these systems the –5V will be present in any event.

The ia7301 uses bipolar PROMs which require only +5V for their operation, so it was decided to generate the –5V supply for the 8080 directly from the +12V supply. The circuit that does this is shown in Figure 5-20; it is a variation of the classic voltage doubler.

A simplified voltage doubler is shown in Figure 5-21. In operation the base of the transistor is pulsed by an oscillator, in our case by the $\phi 2$ clock. When the transistor is turned off, the voltage drop across it rises to V_{CC} which charges capacitor C_1 to V_{CC} through diode D_1. Point A at this time is positive relative to ground which back-biases diode D_2. When the transistor turns on, the side of C_1 tied to the collector is for all practical purposes tied to ground. Point A now is negative relative to ground which back-biases D_1. The second capacitor C_2 is charged through D_2 to a negative voltage relative to ground. In practice the negative charging of C_2 will continue over several cycles; each cycle builds the negative voltage higher until it approaches the absolute value of V_{CC}. This negative voltage can now be used to supply negative biasing requirements for the system; it is easily capable of handling the modest requirements of the 8080. The full circuit shown in Figure 5-20 includes an extra driver transistor and protection diodes to prevent destructive voltages from reaching the 8080 in case of a part failure. The voltage output is set by a zener diode.

KEYBOARD-DISPLAY CIRCUITS

As an educational tool, one of the most important facets of the ia7301 are the keyboard and display circuits. These represent the primary interface between man and machine. The remainder of the I/O ports are used infrequently. The design of the keyboard-display I/O is an excellent example of some of the ways that can be used to eliminate IC logic elements by suitable programming techniques.

As has already been pointed out, the system address decoding is done in such a way that the keyboard/display I/O port will be accessed every time the CXXXH block of memory is addressed. In order to reduce the number of I/O ports required for this part of the system, a scanned technique is used. In this mode, the keyboard is organized as eight rows of three keys each. An output port activates the keys, one row at a time, while an input port searches for key closures. This requires only three bits of the input port since only one row of keys is examined at any one time. The same output port that activates the keyboard is also used to activate the displays.

Each display consists of an 7-segment LED with common anode. This is normally used by tying the common anode to V_{CC} and then selectively providing ground paths for the individual segments by connecting them to 7406 hex inverters. As these inverters are pulled low according to the 7-segment code representing the character to be displayed, the cathodes of the segments to be lighted are connected to ground through the output transistor of the inverter.

In the ia7301 the cathodes of the 7-segment displays are connected to 7406 hex inverters as just described. However, the anodes of the displays, instead of being connected to V_{CC}, are connected to the collectors of transistors that in turn connect them to V_{CC}. By inserting the transistor between the anode of the display and V_{CC} we have added a way to gate the entire display on and off. Eight displays require eight gating transistors. The bases of the transistors are driven by the same output lines that energize the rows of keyboard keys. As each row of keys is activated one of the displays is also activated. Thus the output logic does double duty by scanning both the key-

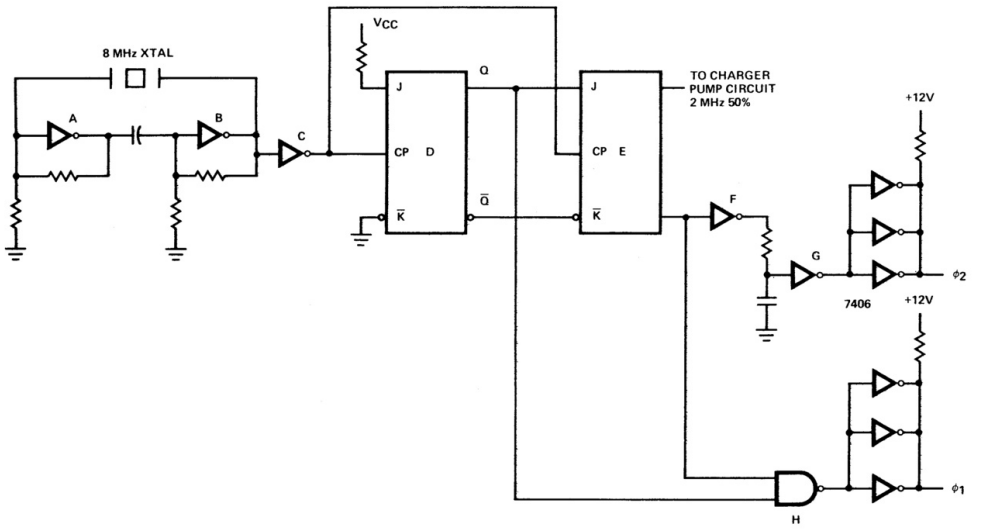


FIG. 5-18 The clock generator for the ia7301 system.

board and the displays at the same time.

By gating the individual displays on and off with discrete transistors the output port driving the 7 segments through 7406 hex inverters can be connected to all of the displays, so that one port and set of inverters drives all eight displays simultaneously. Of course, since only one row of keys and one gating transistor is on at any given time, only one display will be lighted.

The logic that enables the ports is shown in Figure 5-22. The Quad D-type flip-flop that drives the keyboard through the One-of-Eight-Decoder is connected to the A_9, A_{10} and A_{11} address lines. The fourth flip-flop of the Quad is not connected. The flip-flops are clocked by the output of a three-input NAND gate that is connected to the WRITE line and the A_{14} and A_{15} address lines. Thus when A_{14} and A_{15} are high and the WRITE line goes high indicating data is being sent out by the 8080, the output of the gate will go low, providing the falling edge clock pulse the flip-flops need to latch the data on the A_9–A_{11} address lines.

The A_{14} and A_{15} lines effectively address the CXXX block of memory through the NAND gate. Simply addressing the block, and hence the I/O port, is not enough. Somehow data must be transferred, that is, certain rows of keys and displays must be energized. This is done by using the A_9 through A_{11} lines to carry additional data. These lines are connected to the inputs of the flip-flops so that when the WRITE line goes high, the flip-flops latch the data on those address lines into the flip-flops. These are in turn connected to the three low-order inputs of the One-of-Eight Decoder which pulls one of its eight outputs low, corresponding to the code on its inputs.

Each of the outputs of the decoder is connected to a row of the keyboard (see Figure 5-23). As the output goes low, it provides a ground return for any keys that might be depressed. These are tied to V_{CC} through pull-up resistors and blocking diodes so that by reading each of the three keys through the input port the key depressions can be decoded. Without the blocking diodes it would be impossible to determine which key was being depressed since all would present a logic 0 to the input port. The pull-up resistors insure that keys that are not depressed will deliver a true logic 1 to the input port.

Just as the rows of the keys are connected together through the One-of-Eight Decoder, the columns are connected together through the input port. By addressing the CXXX block, one of the rows is selected by the decoder according to the A_9, A_{10} and A_{11} address lines. If there are any key closures in the selected row, that column will present a logic 0 to the input port. Unselected keys will present a logic 1 since they are tied to V_{CC} through pull-up resistors on each column line.

The entire keyboard/display circuit is shown in Figure 5-24. The hex inverters that drive the individual segments of the displays are in turn driven by the outputs of the quad latches. These are clocked flip-flops; a negative-going pulse on the clock pin causes the latches to pick up

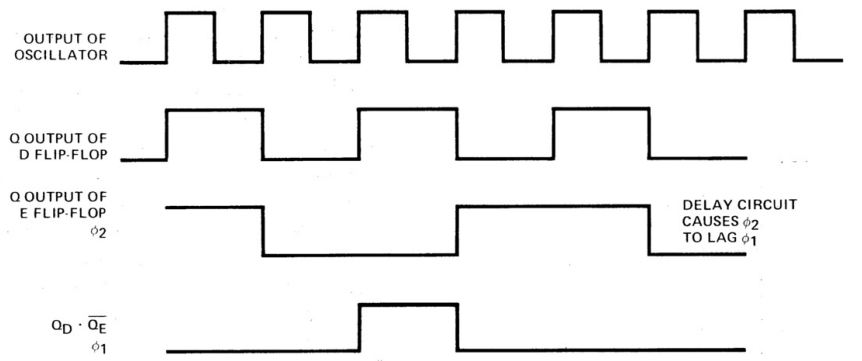


FIG. 5-19 Output waveforms for clock circuit of Fig. 5-18.

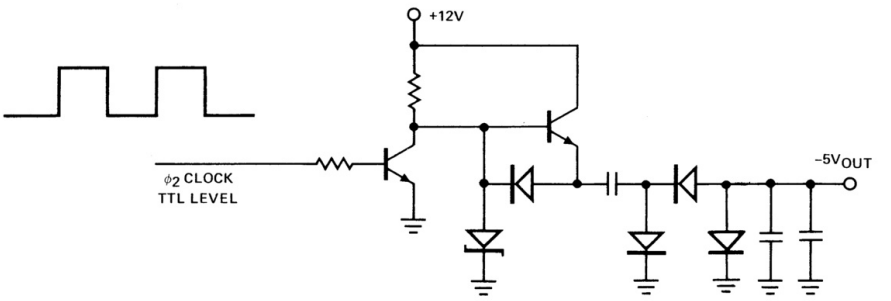


FIG. 5-20 Charge pump used for generating –5V for biasing the 8080.

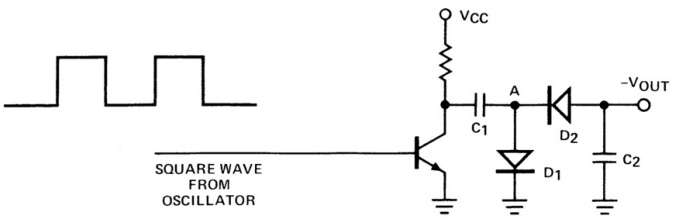


FIG. 5-21 A simplified version of the charge pump used in the ia7301.

and hold the data appearing on the data bus at the time of the clock pulse. The clock pin is tied to the output of the same gate that clocks the quad latch on the A_9, A_{10} and A_{11} pins. This means that whenever the keyboard is enabled selecting one of the rows of keys, the display flip-flops are also clocked, thus latching new data into the flip-flops and causing a new character to be displayed on the LEDs. Of course, even though all of the LEDs receive the same inputs from the hex inverters, only one of the displays will be lighted. This is the one that happens to be connected to the row of keys that is being enabled by the One-of-Eight Decoder; none of the other displays are powered since their gating transistors are turned off.

In practice, the program that drives the displays and keyboards, the system monitor, scans the rows of keys and LEDs so fast that the user is unaware that at any given instant the key he is depressing may not be connected to the CPU. Each row of keys is scanned every millisecond or so; in the brief instant the user depresses and releases the key, the CPU will read that information at least 50 times. This causes a problem with the displays. The same scanning process that covers the keyboard also causes all of the displays to be activated one at a time. The scanning rate is such that all of the displays will appear to be continuously lighted but the duty cycle seen by any individual display is so low that the displays will be very dim. For this reason it is necessary to increase the current through the displays by reducing the size of the current limiting resistor in series with the hex inverters and the display cathodes. When this is done the displays will appear to be normally bright but now there is another problem. If any individual display is on too long the current will be excessive and the display will burn out from the heat generated. Of course, we could be very careful in writing the monitor to be sure that no individual display was on very long before the next keyboard row/display is addressed, but as we shall see in the next chapter, when we actually write the monitor, our hands are full without having to worry about keeping track of the times each LED is on.

The solution is to add a one-shot to the display circuit. This is timed-out with a resistor and a capacitor so the period of the one-shot is about one millisecond. It is triggered by the same gate that clocks the keyboard and display latches. Each time the keyboard is accessed by addressing the CXXX block of memory

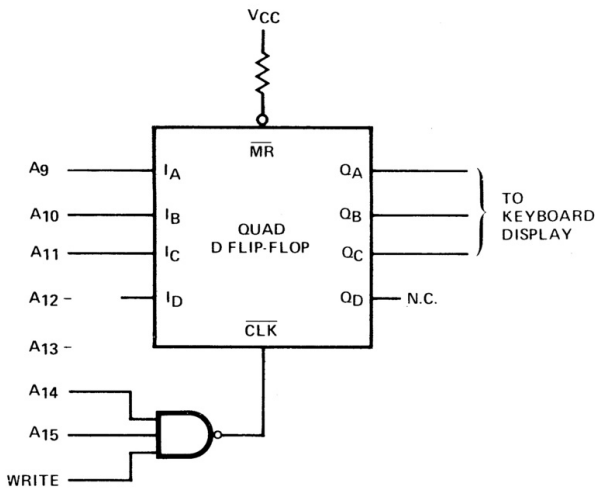


FIG. 5-22 The A_{14}, A_{15} and WRITE lines are gated together to clock the contents of A_9, A_{10} and A_{11} into the D flip-flops.

this one-shot is also triggered. At the conclusion of the one millisecond timing period, the output of the one-shot goes low. This output is tied to the $\overline{MR}$ pin of the display latches. When the one-shot times out the latches are reset which blanks the display. This eliminates the possibility of one of the displays being on too long and burning out, since after one millisecond the display is blanked by the one-shot. The same output of the one-shot is brought back to the input port that is used to read the keyboard columns; three bits of the input port are for the keyboard columns and one is for the display one-shot. This allows the user to monitor the output of the one-shot, which is useful for insuring that all of the displays are operating with the same duty cycle and hence are equally bright.

The latch that controls the One-of-Eight Decoder which in turn drives the keyboard rows is clocked whenever the CXXX block of memory is accessed with a memory write operation. The data that is loaded into the 4-bit latch is the content of the A_9, A_{10} and A_{11} address lines. This will remain in the latch until the next time the CXXX is addressed with a memory write instruction. Thus one of the keyboard rows and one of the display driver transistors will be activated until it is changed by another write to CXXX. Once a row has been activated, the contents of the display latches will appear in the LED that has been activated. This is done automatically whenever a row of keys is activated. If we wish the display to appear blank, we must load the latches with all logic ones.

After activating the row of keys, the input port must be read. This port does not contain a latch, so data must be read "on the run." Instead of a latch, a pair of quad two-input multiplexers are used. In one mode the data from the keyboard, the display one-shot and several other internal signals are routed through the multiplexer. This multiplexer contains an output enable pin which we have tied to $\overline{A_{14}}$. When this line is low, meaning A_{14} is high, the outputs of the multiplexer are placed onto the data bus and read by the CPU. The direction of the multiplexer is controlled by the A_{15} line. When this line is high the multiplexer inputs tied to the keyboard and display one-shot are active. Thus a read to CXXX will read the keyboard and other internal signals into the 8080. When the select input of the multiplexer tied to A_{15} is low, the other multiplexer inputs are active. These are tied to the external input port and are available to the user.

To summarize, the keyboard rows and the digit displays are both activated by performing a memory write operation to the CXXX block of memory. This occurs regardless of what we place on the data bus since the keyboard row and LED digit are selected on the basis of the A_9, A_{10} and A_{11} address lines. Addressing C2XX selects row and digit 0, C4XX selects row and digit 1, C6XX selects row and digit 2, etc. This being so, we are free to use the data bus to carry other information at the very time we are using the address lines to select one of the keyboard rows and one of the digit displays. At the same time that the latches that drive the decoder circuits used to select keyboard row and display digit, the output latches that drive the segments of the displays are clocked. Thus the data on the data bus at that time is latched into the flip-flops and used to drive the displays; one instruction both selects the display and keyboard row and loads the display with the digit to be displayed.

Since there is only one output display port it is up to the programmer to be sure that as the keyboard is scanned the display port is continually loaded with the correct digit that corresponds to the digit display that is activated. Each must be followed by a memory read operation in order to check for key closures. The arrangement of keys within the keyboard was carefully chosen to make it easy to

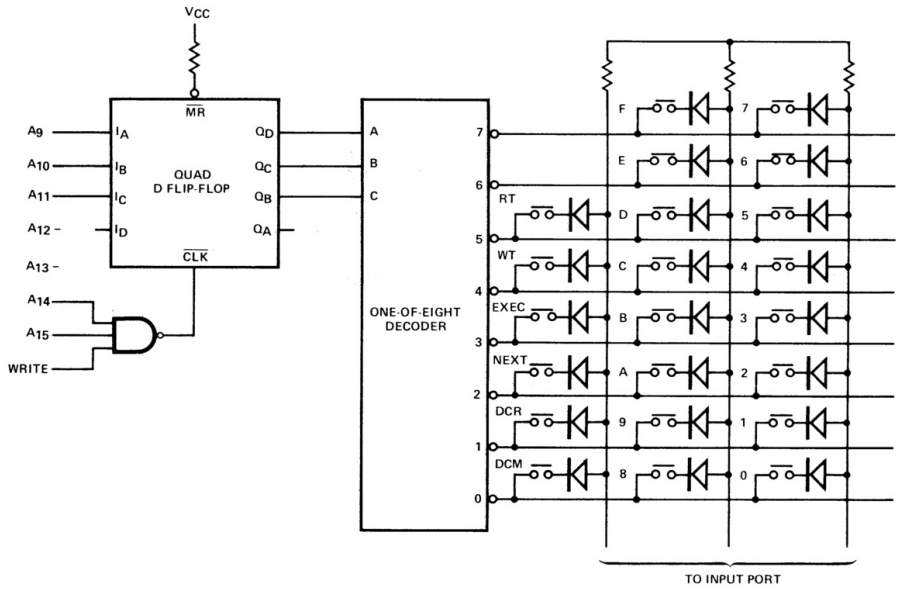


FIG. 5-23 The quad flip-flop drives the keyboard through a one-of-eight decoder. The decoder selects one row at a time while the input port checks for key closures in that row.

determine which row of keys is activated. Thus key #0 is in row #0, key #1 is in row #1, key #2 is in row #2, etc. This will be covered in detail in the next chapter when the problem of writing the system monitor is brought up.

As has already been pointed out, the external input port is brought into the system through the two quad input multiplexers. With the direction pin of these devices in one state the internal input port interfacing to the keyboard and the display one-shot is read. With the direction pin in the other state the external input port that is available at the edge connector is read. There is also an external output port made up of a pair of clocked quad latches. The information on the data bus is latched into these flip-flops when the clock input is pulsed. This occurs whenever the WRITE line, the A_{12} and the A_{14} line all go high. This coincidence is detected by a three-input NAND gate whenever a memory write to address 5XXX occurs.

TAPE RECORDER INTERFACE

When the user must leave the system and remove the power supply the contents of the RAM memory will be lost. Since it is the RAMs that contain the user's program, all of his hard work will be lost. Of course, he could write the program down as he perfected it, but a long program requires some time to key in and mistakes are easy to make. The tape recorder interface allows him to load the entire contents of RAM memory onto a cassette tape by using the tape recorder interface. Any recorder may be used that has a remote microphone plug and an earphone or external speaker plug. These are connected to the edge connector of the system through cables supplied for this purpose. When the recorder is placed in the record mode and the tape put into motion, pushing the WRITE TAPE button on the keyboard will cause the contents of the RAM memory to be loaded onto the tape. The exact format used to record on tape will be covered in the next chapter.

When it is desired to recover the program the tape must be rewound, placed in the playback mode and once again the tape set in motion. When the user presses the READ TAPE key, the information contained on the tape will be read back into the system until the entire RAM memory has been loaded with the contents of the tape. To ease the problem of starting the tape and then pushing a key, the program that writes the data onto the tape precedes the date with a 30-second leader. The process of loading or reading the tape requires about 2 minutes including the 30 seconds of leader.

The circuits that accomplish this are fairly simple. An output port is used to write the signals onto the tape and to simultaneously drive the discrete LEDs that provide a monitor on the operation of the system. For this purpose only 4 bits are needed, one for the tape writing

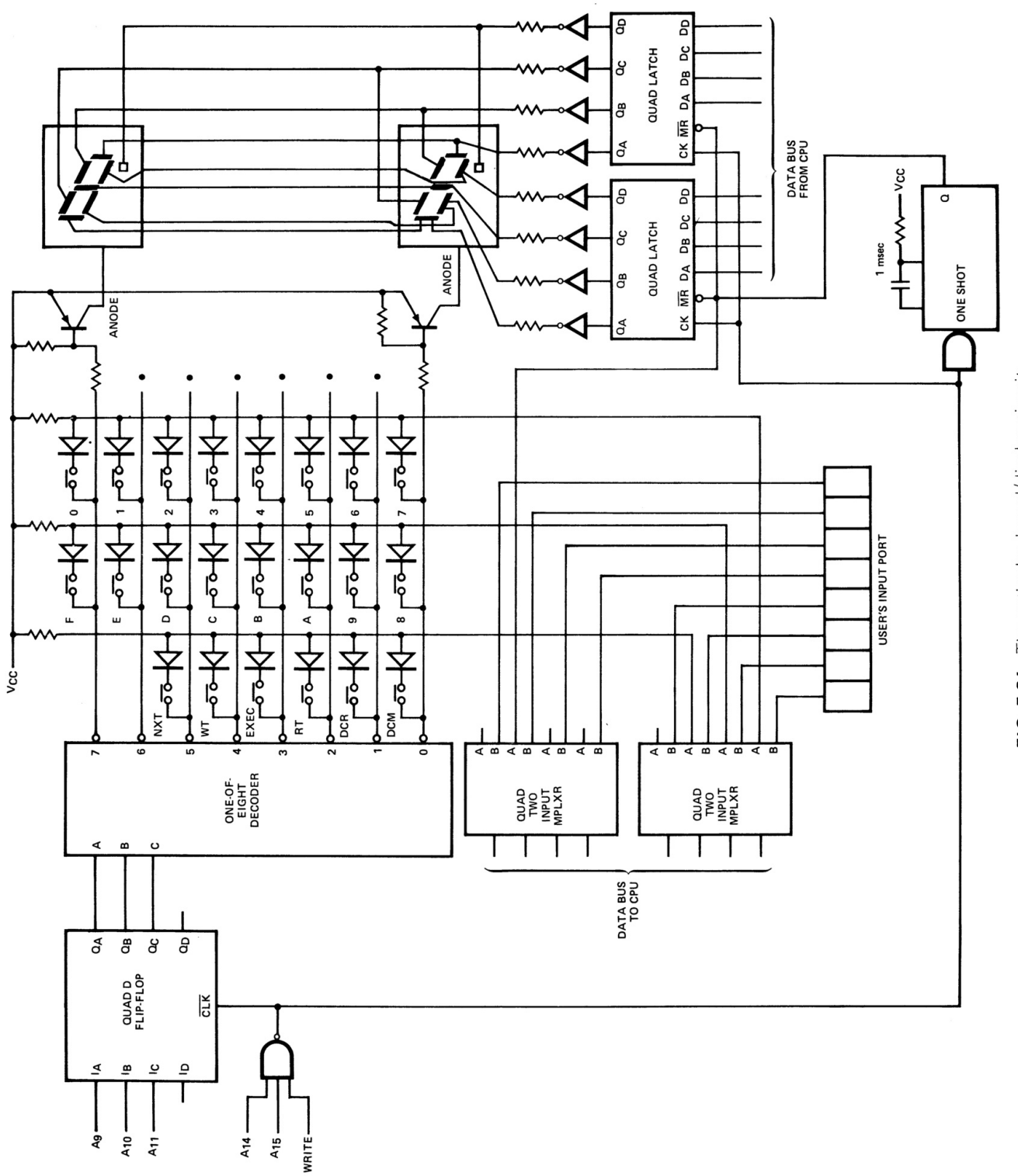


FIG. 5-24 The entire keyboard/display circuit.

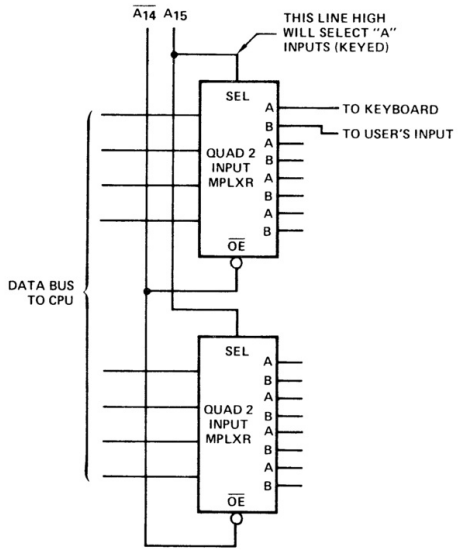


FIG. 5-25 The user's input port and the internal input (keyboard, etc.), are both read through a pair of quad multiplexers. The SEL input is tied to A_{15} and is used to select which of the two ports will be read into the 8080. The OE input enables the tri-state outputs.

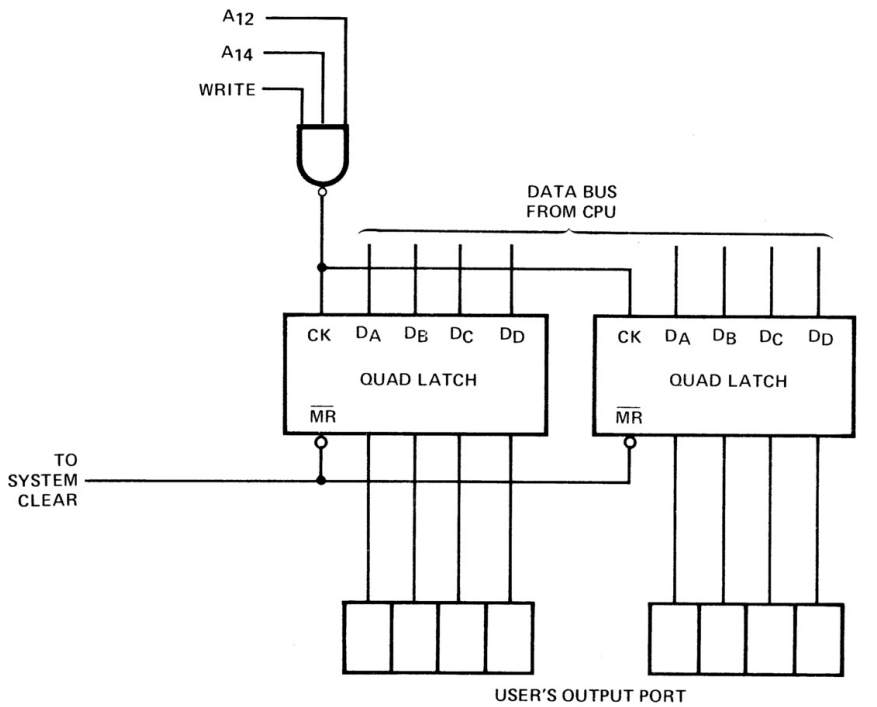


FIG. 5-26 The user's output port is driven by a pair of quad latches clocked by coincidence on A_{12}, A_{14} and WRITE.

signal and the other three to drive LEDs. Each time a memory write is sent to this port the LEDs and the tape write circuit are activated. Since the port contains a quad latch, it will retain the output information until it is changed by a new memory write. The tape head is coupled to the computer with DC circuits; no coupling capacitors are used. This means that the flip-flop that drives the record head must be toggled in order to record a tone on the tape. The basic format calls for two tones or square waves to be recorded for each bit. Depending on whether the signals are tone A for 1/3 of the time and tone B for 2/3 of the time, or vice versa, determines whether the bit will be interpreted as a 0 or a 1. Needless to say, the tape read program must be written so that the two tones can be detected and their recording times determined.

There are three LEDs driven by this same port. One is used to indicate when the user's program is being executed, one for the WRITE TAPE mode and one for the READ TAPE mode. The other modes are very obvious to the user since they each place a mode symbol on the digit displays. This is true for all the modes except EXECUTE, READ TAPE and WRITE TAPE. Since the outputs of the port latches are not capable of driving the LEDs directly they are first buffered by open collector hex buffers. Because of the modes the LEDs are to indicate, there can never be more than one on at once. Thus to write a signal on the tape, a series of memory write operations to the output port must take place. The flip-flops connected to the EXECUTE and READ TAPE LEDs must receive 0's. After the buffer inverts these signals they will be 1's which will cause those two LEDs to be blanked. The flip-flop connected to the WRITE TAPE LED must receive a logic 1 in order to light it. This is the second bit position, so sending 03H out on the data bus to this port will both light the LED and write a signal onto the tape. Of course, since the tape must receive a square wave, the 03H signal must be followed by a 02H signal. This returns the tape head to a low voltage representing the valley of the square waves; the LED remains lighted.

Reading the tape requires only an AC coupled transistor amplifier with its output fed into the same internal input port that reads the keyboard and display one-shot. This represents the fifth bit of that port. The program for reading the tape is fairly involved and will be covered in the next chapter.

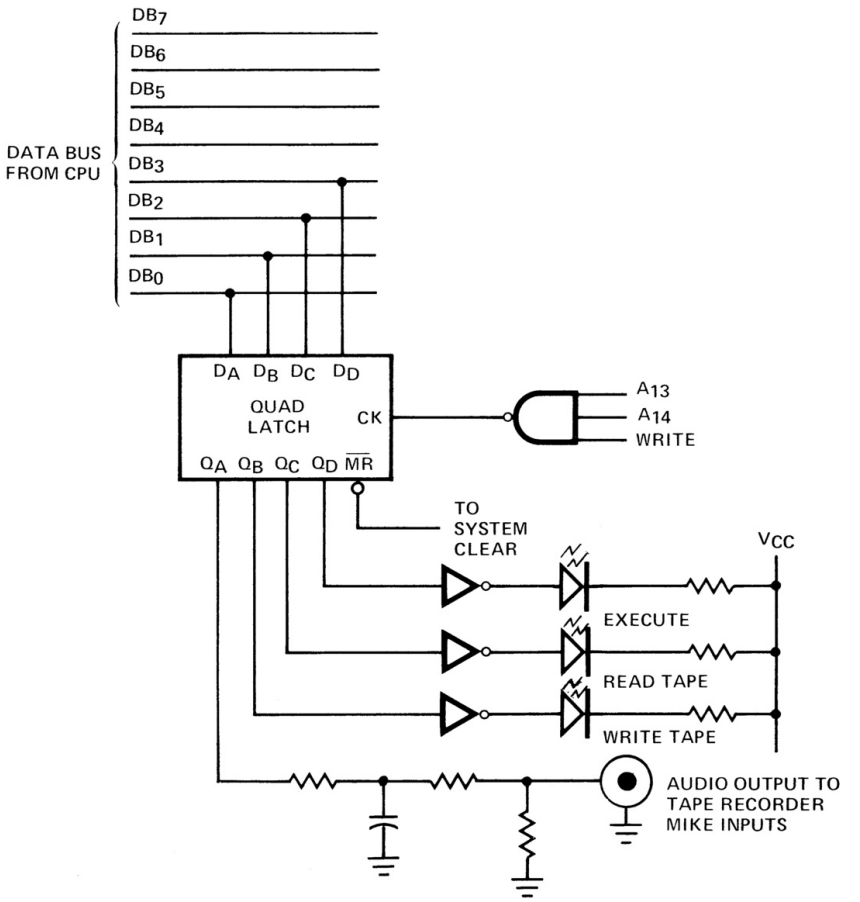


FIG. 5-27 Tape recorder WRITE circuit and LED monitors.

EXECUTE PROGRAM MODE

Although a detailed discussion of the workings of the ia7301 appears in the next chapter, we cannot intelligently discuss the hardware aspects of the STEP mode without understanding a little about how the system operates under program control during the EXECUTE and STEP modes. A much more detailed description will be given in Chapter 6.

As has been pointed out on numerous occasions throughout this book, when the ia7301 is first energized it comes up in the system monitor. That is, it is executing the program permanently contained in PROMs even though those devices are not the low address ports of the memory. While in the system monitor, the user writes his program using the keyboard and the DCM mode. That program must have a starting address usually chosen to be location 100 (locations 0–99 are reserved for use by the monitor and conflicts may occur if the user tried to write his program into those locations). During the process of writing the program the system stores the instructions as keyed in by the user into RAM locations as directed by the user. To execute his program it is only necessary to somehow load the 8080 program counter with the starting address of the user's program and the system will begin to execute that program exactly as written.

While debugging programs it is frequently desirable to be able to examine the various registers of the 8080, the accumulator, the stack pointer, etc. This is done by using the DCR mode of the system monitor. In this mode the registers of the user's program are displayed on the LEDs and may be modified by the user. This is actually done by selecting certain low-order RAM locations to act as the images of the CPU registers when the system is executing the user's program. Thus if the user stops the system while executing his program, the monitor causes all of the CPU registers, including the program counter, to be saved in the RAM locations reserved for them. There they are easily accessible and can be examined and modified by the user via the DCR mode. Once he is satisfied, the user pushes the EXECUTE button and the monitor causes the images of the registers, which the user may have now modified, to be loaded back into their correct places in the CPU. When the program counter is loaded with its image the system control is transferred to the user's program and execution picks up exactly where it was interrupted. When trying a new program for the first time, none of the registers will have been preconditioned (the user's program itself should do that). The program can be executed by pushing DCR and loading the program counter image in RAM with the address of the first instruction of the user's program. When EXECUTE is pushed, the new program will be executed beginning at the first instruction.

SINGLE-STEP MODE

One of the most useful features of the ia7301 is the ability of the user to single-step through his program, executing only one instruction and then entering an idle state while the address and instruction to be executed next are displayed. This allows him to follow a program step-by-step. By using the DCR mode the working registers of the 8080 can be examined and changed to debug the program.

The hardware implementation of this feature involves the interrupt capability of the 8080. This is the only interrupt that is allowed in the ia7301. The timing requirements on the interrupt are fairly

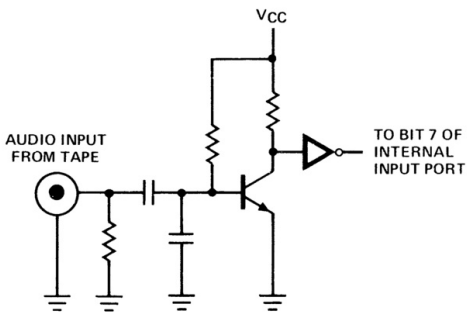


FIG. 5-28 Tape recorder READ circuit.

tight and must be synced to the status word that appears on the data bus at the beginning of every machine cycle. This is done by generating a $\overline{\text{STROBE}}$ signal by NANDing the SYNC and $\phi 1$ clock signals together. Coincidence of these two signals only occurs during the T_1 state when the status word is on the data bus. The $\overline{\text{STROBE}}$ signal thus generated is used in several circuits in the interrupt and single-step parts of the system.

The STEP key is connected to the inputs of the single-step one-shot. This is a re-triggerable type that fires on the rising edge of the trigger pulse. The trigger pulse itself is the output of an OR gate contained within the one-shot IC. One input of the gate is a positive true input and the other is a negative true input which is tied to the $\overline{\text{STROBE}}$ line. It is the output of the OR gate that triggers the one-shot; a rising edge here will cause the device to fire. The positive true input is connected to one side of the STEP key and to a pull-up resistor to V_{CC}. The other side of the key is grounded. Thus in the normal state of affairs with the key being open, the one-shot OR gate receives a logic 1 signal on its positive true input. This has the effect of holding the output of the OR gate high so that no trigger can take place. The $\overline{\text{STROBE}}$ line going low during every T_1 has no effect on the output of the gate.

When the STEP key is depressed the positive input is pulled to ground through the key and it becomes the $\overline{\text{STROBE}}$ line that controls the output of the OR gate. As this line goes high on the negative true input it causes the output to go low. When the $\overline{\text{STROBE}}$ goes low the output of the OR gate will go high, presenting the necessary rising edge to the one-shot. It fires and the output goes high for the first time.

The output of the one-shot is tied to two other places within the system. First it is tied to the input of a three-input NAND gate. One of the other outputs of that gate is tied to $\overline{A_{15}}$ which means that the input will be low whenever the system is executing out of the system monitor. The output of the gate is the $\overline{\text{STEP}}$ signal, active low. When the monitor is active the output of this gate will be high and the signal disabled. The one-shot is also tied to one of the inputs of the internal input port. This is the sixth bit of that port and it is read every time the keyboard is read which is very frequently since this is the mode the system is in most of the time.

When the system detects a pulse from the step one-shot it enters the single-step mode. In this mode the monitor accesses the RAM locations that are being used to hold the images of the user's program counter, stack pointer, accumulator and other working registers. These are loaded into the 8080 in their appropriate spots and execution of the user's program begins. As soon as this happens, the $\overline{A_{15}}$ line goes high and the NAND gate connected to the one-shot is free to gate the data on its inputs. Up to now the system has been executing the system monitor which requires the $\overline{A_{15}}$ line be low, which in turn forces the output of the three-input NAND gate high. But as soon as the monitor reloads the 8080 with the data from the user's program and that program begins to execute, it will be working out of RAM and A_{15} will be high. The other two inputs of the NAND gate go to the output of the step one-shot and DB_5, the sixth line of the data bus. Coincidence is required on all three in order to produce an active $\overline{\text{STEP}}$ output.

When the program control is transferred to the user's program in RAM the first instruction of that program as indicated by the user's program counter at the time the program was entered will be fetched by the 8080. As this happens the status word placed on the data bus will indicate the cycle is the M_1 cycle, meaning it is the first cycle of an instruction fetch. DB_5 is high to indicate M_1 and the step one-shot is still high from the original depression of the STEP key. Thus all three inputs of the NAND gate are high

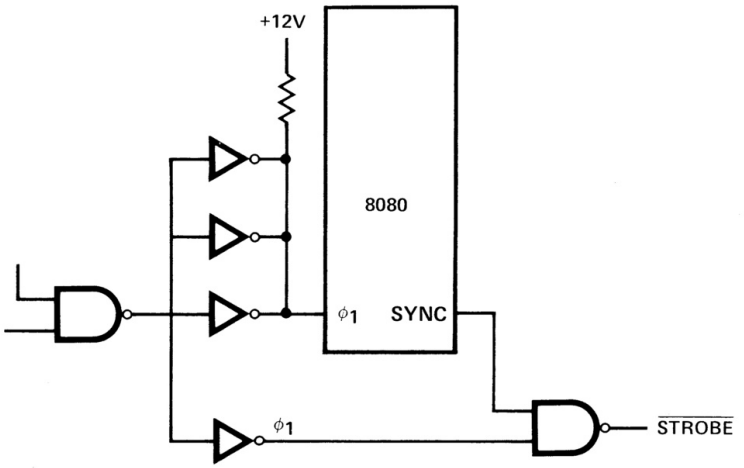


FIG. 5-29 ϕ_1 clock and sync are NANDed together to produce STROBE. This signal is low whenever the status word is on the data bus.

and its output, $\overline{\text{STEP}}$, will be low. This signal is applied to the $\overline{K}$ input of a flip-flop clocked by STROBE. The J$\overline{K}$ flip-flop used is clocked by the rising edge of the clock pulse. At the time the NAND gate delivers the $\overline{\text{STEP}}$ signal to the flip-flop the $\overline{\text{STROBE}}$ line is low since the status word is on the data bus. Shortly before the status word is removed, $\overline{\text{STROBE}}$ will go high which will clock the flip-flop. Since the J input of that flip-flop is wired to ground, the clocking signal will result in the $\overline{Q}$ ouput going high. It in turn is wired to the interrupt input of the 8080 which will respond by causing an interrupt. That interrupt will not occur, however, until the instruction being executed at that time is completed. This instruction is the first instruction being executed out of the user's program. No sooner is it completed then an interrupt occurs.

The 8080 responds to the interrupt being presented on its interrupt input by sending out the interrupt acknowledge signal in its status word during the next instruction cycle. This is done by placing a logic 1 on DB_0. We pick this but off the data bus after it has been inverted by the output bus buffers. At this point it is a logic 0 and we use that bit to condition the inputs of another J$\overline{K}$ flip-flop. This flip-flop is also clocked by $\overline{\text{STROBE}}$ so that it only responds to the status word. The Q output of this flip-flop is NANDed together with DB_{IN} and then inverted to disable the input bus gates that drive the data bus inputs to the 8080. Thus when the 8080 acknowledges the interrupt and reads the data bus to determine its restart vector, it finds all 1's on the bus. This is the instruction code for RESTART 7, and the 8080 performs a jump to the location corresponding to RESTART 7 which is location 0038H in RAM. As part of the initialization process that occurs whenever the monitor is fired up, that location contains a JUMP instruction to the STEP mode routine in system monitor.

When the STEP mode in the monitor is entered, the displays are loaded with the address contained in the user's program counter and the contents of that location. The system then idles, waiting for the next depression of the STEP key.

In summary, the effect to the user is that while executing a program that seems to be hung up, he pushed the STEP key. The system goes into an idle condition while the address and contents of the next instruction to be executed are displayed on the LEDs. The user can change that instruction if he wishes by pushing DCM, punching in the address of the instruction and the replacement for the instruction there. He can then examine the registers by pushing DCR and modify

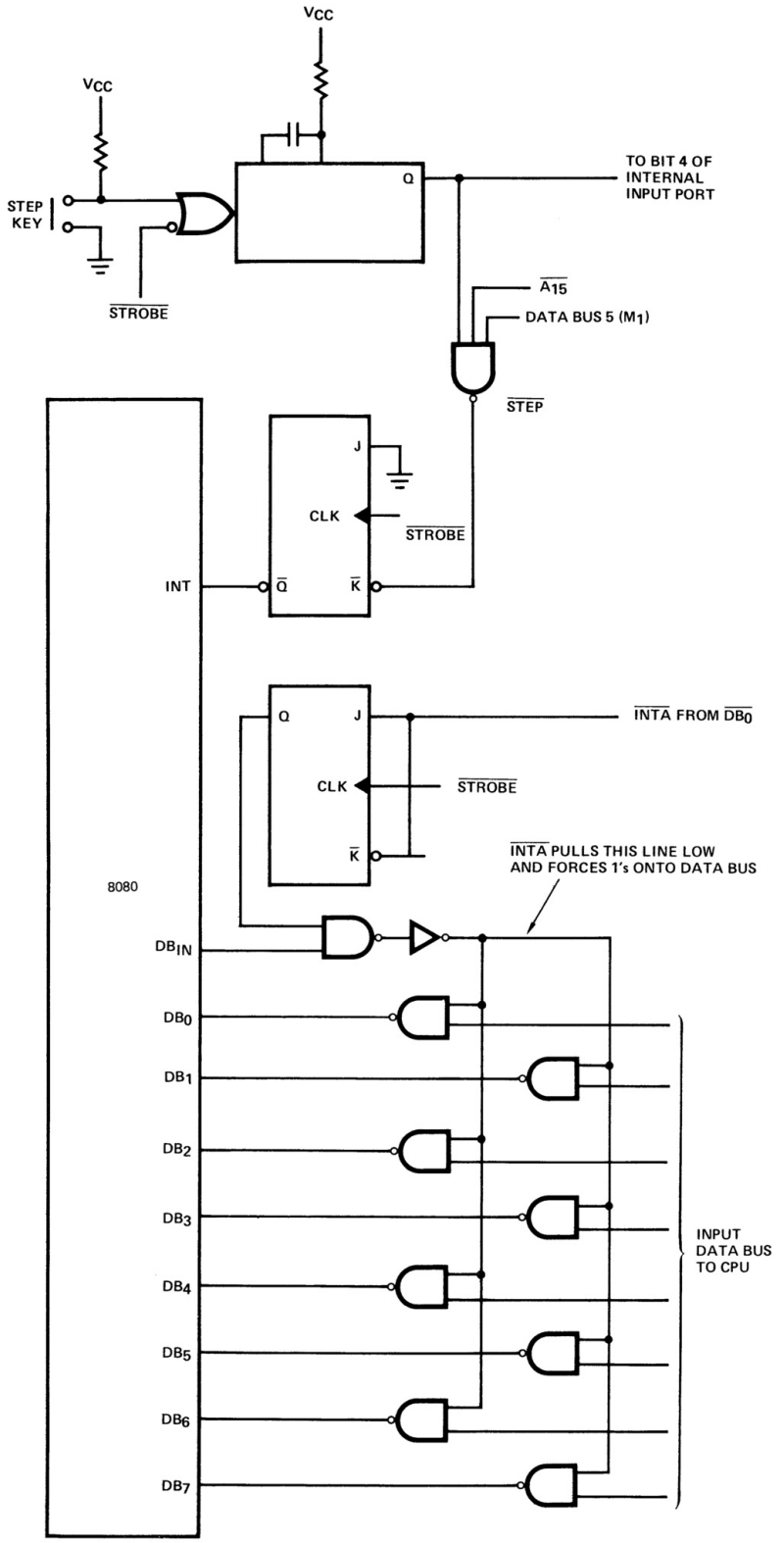


FIG. 5-30 STEP and INTERRUPT circuits.

them. As soon as he pushes STEP, the user's program is begun at the address contained in the RAM image of the user's program counter. One instruction is executed and then control returned to the monitor. The system once again idles, displaying the address and contents of the next instruction to be executed. In this manner the user's entire program may be executed, one instruction at a time, until the difficulty is located.

In this chapter we have seen some of the problems and trade-offs associated with the hardware design of an operating microcomputer system. In the next chapter we will go through the process of programming it with a system monitor. It is suggested that if you are unfamiliar with the instruction set of the 8080 microprocessor you turn to Appendix C in the back of the book and review the instructions there before proceeding to Chapter 6.

Chapter Six
Writing the Monitor Program

Now that the hardware aspects of the system design have been completed, it is time to move on to the monitor program. This will provide the system with the intelligence it will need to respond correctly to our commands as entered through the keyboard. The monitor program comes up when the system is first energized so that from the user's standpoint, the system is immediately able to respond to the keyboard. This program is an excellent example of a microcomputer program since the range of problems that it must cope with is extremely wide.

The assumption is made here that the reader is already familiar with the instruction set of the 8080 microprocessor. If this is not true, the reader should review the instruction set in Appendix C in the back of this book. The programming examples in this chapter are all presented in the format of assembly language programming. In most cases the meaning should be immediately clear to the reader. Where this is not true, additional explanation will be given.

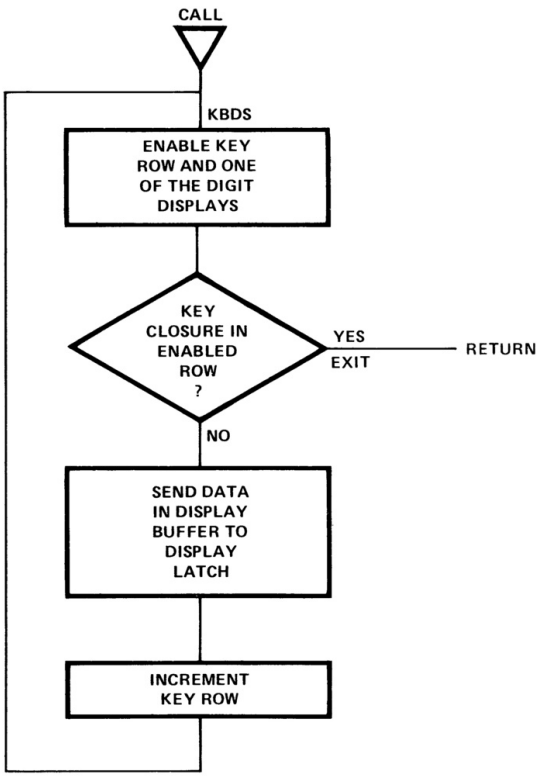


FIG. 6-1 A simplified flow diagram of KBDS. The same IC is used to enable both a keyboard row and one of the LED displays. Reading the internal input port then checks for a key closure. Writing data to the same port address will cause the data to appear in the enabled display. The IC decoder is then incremented to enable the next row of keys and LED digit. KBDS is a callable subroutine. It is exited, when a key closure is found, by executing a return.

INTRODUCTION

The writing of any program, especially one as complex as this, must begin with an idea of how the program is to flow as a whole. This may be done with flow diagrams or, in simpler programs, just a mental idea of how the thing will hang together. Without this picture, the programmer will wander from one point to another, having to rewrite program segments often as conflicts between them become apparent.

In the case of the ia7301 monitor, the picture is relatively simple. The system is visualized as always being in one of two modes, either executing the user's program or executing the monitor. Within the monitor is a callable subroutine called KBDS, Keyboard Display, which results in the system doing a continuous loop displaying data and searching for key closures in the keyboard. In reality, this is the mode the system will spend most of its time in.

The monitor causes eight RAM locations to be selected as display buffers and eight more to be selected as display registers. During the KBDS subroutine the contents of the eight display buffers will be sent one by one to the output display latches. As the subroutine scans the keyboard rows each of the displays is enabled, one at a time. The enabled 7-segment display lights up with the contents of the display latches. The next key row is then enabled which also causes the next display to be enabled. Part of the KBDS subroutine is responsible for loading the display latches with the contents of the next display buffer so that the correct number will appear in the appropriate display. At the same time the program is scanning the displays, it is also scanning the keyboard. When a valid key closure is found the subroutine is exited with the image of the key in the accumulator. See Fig. 6-1.

At this point it is necessary to determine whether the key is one of the mode keys or one of the 16 hexadecimal keys. A special subroutine, called MDCCK, Mode Change Check, performs this func-

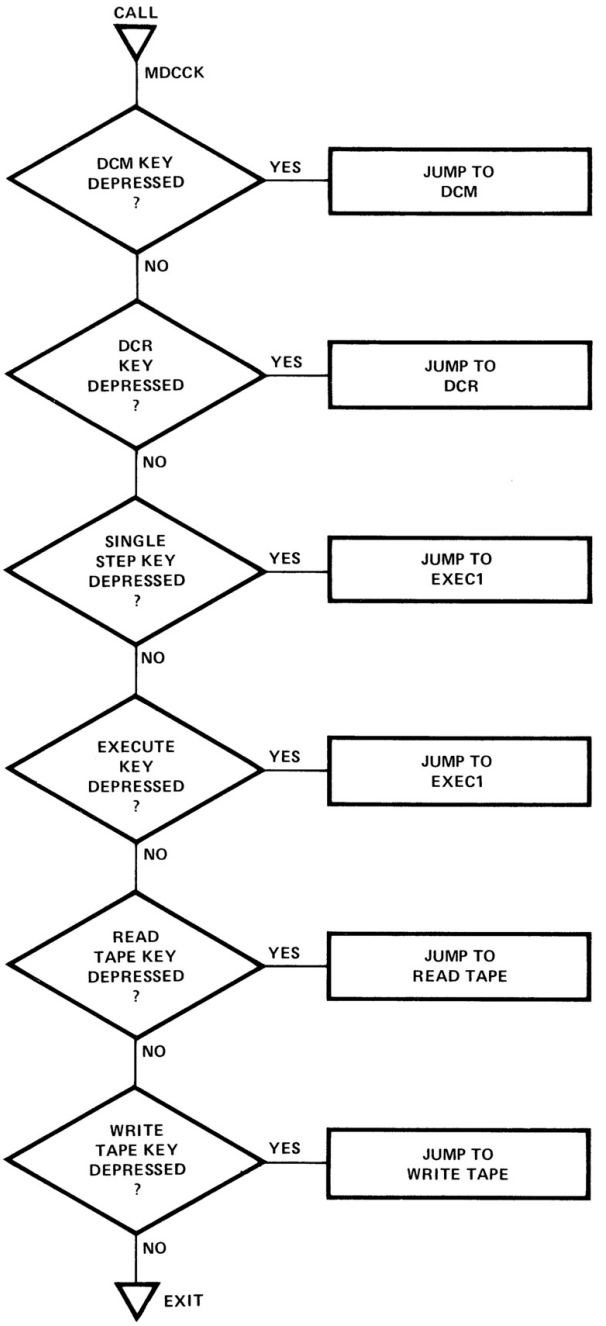


FIG. 6-2 A simplified flow diagram of MDCCK. This routine checks to see if any of the mode keys are depressed. If so, a jump to the appropriate mode routine occurs. Both STEP and EXECUTE keys cause a jump to the EXECUTE mode. STEP, however, causes an interrupt so that only one instruction of the users program is executed. EXECUTE also jumps to the user's program via the EXECUTE mode but now the program free-runs. If none of the six mode keys are depressed, the key closure must have been from the NEXT key or one of the hexadecimal keys. In this case the subroutine is exited by a return with the key image in the accumulator.

tion. If one of the mode keys has been depressed, MDCCK is exited by a jump to the appropriate mode program. If not, the MDCCK routine is exited by a return to the calling program with the hex number contained in the accumulator. See Fig. 6-2.

Most of the operations within the microcomputer system are done with binary numbers. These, however, cannot be displayed directly on the 7-segment LEDs without doing a code conversion on them. This is done by a third callable subroutine, CONV, which takes the contents of the display registers one by one and performs a code conversion on them. The resulting code is loaded into the display buffers. The three subroutines, KBDS, MDCCK, and CONV, are intended to be used together in succession. They could be combined into one subroutine and thus shorten the amount of PROM memory required by several bytes for each CALL and RET instructions. By keeping them separate, however, they are more versatile and can be called by the user in his own program. This allows him to use the mode keys for his own purposes without causing an automatic jump to a mode program segment. See Fig. 6-3.

When the system is first energized the monitor is entered, the system is initialized and the DCM mode is entered. In this mode the user is able to access memory locations of his own choosing and if he so desires, to modify them. Once the initial address has been entered, it is only necessary to depress the "NEXT" key to increment the address. As each address appears, in the four left-most displays, the contents of that memory location appear in the two right-most displays.

During the DCM mode we can see a typical use of the KBDS, MDCCK and CONV subroutines. Once the display registers have been loaded with the correct values, CONV is called to perform the 7-segment code conversions on the display registers and load the display buffers with the 7-segment equivalents of the display registers. The KBDS subroutine is then called to display the data and search for a key closure. The system will loop in the KBDS routine until a key closure is found. Since even a long program will execute in a few milliseconds, while the user may examine the displays for several, perhaps even many seconds it is obvious that the system will spend the bulk of its time in KBDS waiting for the user to hit a key. As soon as this happens the key is checked to be sure the user is not depressing one

of the other mode keys, meaning a switch to another mode should occur. Assuming the key is a number key, a return to the calling program is executed and the main program continues with the data presented by the key closure contained in the accumulator.

In addition to the RAM locations reserved for the display buffers and registers, other locations are set aside to act as the image registers of the various CPU registers. Thus, one of the RAM locations is used to hold the image of the accumulator, another for the image of the flag bits, another for the B register, and so on. In this way, each of the CPU registers, from accumulator to stack pointer, is duplicated in RAM locations reserved for that purpose. This allows the system to switch back and forth between the monitor and the user's program without losing the contents of any of the working registers. Of course, they could be simply saved on the stack but this is somewhat more difficult since a requirement of the DCR mode is that each of the registers is displayed and made available for change. Thus when the EXECUTE key is depressed the monitor retrieves each of the register images from the RAM and loads them into the appropriate registers in the CPU. When the STEP key is depressed, the registers are stored in their image registers in RAM and the current address and instruction displayed.

The KBDS can thus be seen as the subroutine that ties the entire monitor together. In this vein MDCCK and CONV are used with KBDS to perform the function of getting key images, directing the displays and causing jumps to the other mode routines.

We will now begin the process of actually writing the monitor. If this seems to be somewhat disjointed, that is the way that programs are actually written. Individual problems are identified and subroutines written to cope with them. At all times care must be exercised to be sure that the subroutines are consistent with the overall picture of the complete program. During this process there will be much backtracking and rewriting as more efficient attacks are found and implemented. As a guideline during this process, it should be kept in mind that the problems identified should be kept as small as possible. The tendency of the beginning programmer is to look at problems that are far too large to be treated efficiently. If these large problems are broken down into much smaller units, the solutions will be found much quicker. The solutions to the small problems are then put together to solve the larger ones. This has the advantage that many of the small solutions will be treated as callable subroutines that can be used by other portions of the program with a resulting savings in memory.

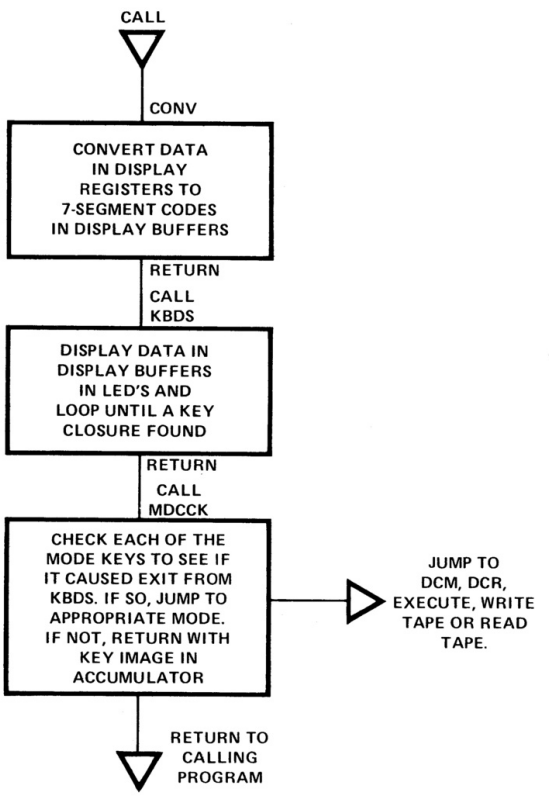

CALL
CONV
CONVERT DATA IN DISPLAY REGISTERS TO 7-SEGMENT CODES IN DISPLAY BUFFERS
RETURN
CALL KBDS
DISPLAY DATA IN DISPLAY BUFFERS IN LED'S AND LOOP UNTIL A KEY CLOSURE FOUND
RETURN
CALL MDCCK
CHECK EACH OF THE MODE KEYS TO SEE IF IT CAUSED EXIT FROM KBDS. IF SO, JUMP TO APPROPRIATE MODE. IF NOT, RETURN WITH KEY IMAGE IN ACCUMULATOR
JUMP TO DCM, DCR, EXECUTE, WRITE TAPE OR READ TAPE.
RETURN TO CALLING PROGRAM

FIG. 6-3 The CONV, KBDS and MDCCK subroutines are designed to be used in sequence. In this combination, the contents of the display registers appear in the LED's. When a key is depressed the subroutines are exited via a jump to the mode or a return to the calling program depending on whether a mode key or a hexadecimal key is depressed.

THE FIRST SUBROUTINE

Before getting into the full-blown program segments like KBDS, let's try a short subroutine to warm up. This routine will be used to blank all of the displays before entering the modes that use them. If this were not done, many of the digits would contain random numbers that represented the contents of RAM locations that were never initialized. To prevent these confusing digits from appearing we will call this subroutine before KBDS. An option of the subroutine allows the user to load all of the displays with one character rather than blanking them.

We begin by loading the accumulator with all ones. The contents of the accumulator can then be moved to each of the

display registers in turn. Since logic ones in the display registers blank the segments of the display, loading them with all ones will blank all of the segments. When each of the display registers is loaded in this manner, the entire display will be blanked. Of course, to move the contents of the accumulator or any other CPU register into a memory location, some way must be found to specify the address of the memory location to receive the new data. That way is to load the H and L register pair with the address of the memory location in an LXI instruction. This obviously destroys the prior contents of the H and L registers which may cause a problem. So, before loading the H and L registers by the LXI instruction, we will first PUSH H onto the stack so that the contents of the H/L registers can be retrieved later. Since there are eight display registers the contents of the accumulator will have to be moved to each of them in turn. This means that the H/L registers pointing to the first display will have to be changed seven times so that each of the display registers will receive their data.

The display registers might appear anywhere within the RAM portion of the system memory. An LXI instruction can then be used to load the H/L registers with the addresses of each of the registers, eight registers—eight LXI instructions. But an LXI instruction requires three bytes of memory, one for the instruction and two for the data that is to be loaded into the register pair. If we use a little care in the selection of RAM locations for the display registers this can be shortened considerably. The most obvious choice for the addresses of the display registers is eight contiguous locations. Then to access the next display register all that is necessary is to increment or decrement the contents of the H/L registers. This has the same effect as loading a new address into the register pair but only requires one byte of program memory.

We could choose the display registers to occupy any location in RAM memory as long as the eight were contiguous. There is one area of the RAM that is particularly convenient and placing the display registers here will not interfere with the user's program. The area is at the very bottom of the memory, that is the first locations in memory. This area tends not to get used for storing user's programs since it is broken up in one or more places by the RESTART instructions that must appear there in order to service the interrupts that may occur. In the ia7301 there is only one RST instruction at location 38H. The locations below this address tend never be used since the programmer must worry about running his program into the RST at location 38H. This means the locations can be used by the monitor for the display registers, buffers and image registers without worrying about reducing the amount of memory available to the user.

In Fig. 6-4 is the memory map of the lower memory locations. Here we can see the eight display buffers assigned locations 0—7H. It should be noted that the process of assigning the buffers to these locations does not imply any physical action or hardware changes; the only implications of the assignments are in the way the monitor program uses them to store the 7-segment codes that will be sent to the displays. While any RAM location can be used for multiple purposes, that is, for storing different numbers at different times, this is not true of the display buffer locations. While it is conceivable that a clever programmer could use them for temporary storage of

ADDRESS	CONTENTS
1B	PC IMAGE (HY-BYTE) PCHIM
1A	PC IMAGE (LO-BYTE) PCLIM
19	SP IMAGE (HY-BYTE) SPHIM
18	SP IMAGE (LO-BYTE) SPLIM
17	H IMAGE HIMG
16	L IMAGE LIMG
15	E IMAGE EIMG
14	D IMAGE DIMG
13	C IMAGE CIMG
12	B IMAGE BIMG
11	FLAG IMAGE FGIMG
10	ACCUMULATOR IMAGE ACIMG
0F	DISPLAY REGISTER #7 DIRG7
0E	DISPLAY REGISTER #6 DIRG6
0D	DISPLAY REGISTER #5 DIRG5
0C	DISPLAY REGISTER #4 DIRG4
0B	DISPLAY REGISTER #3 DIRG3
0A	DISPLAY REGISTER #2 DIRG2
09	DISPLAY REGISTER #1 DIRG1
08	DISPLAY REGISTER #0 DIRG0
07	DISPLAY BUFFER #7 DISP7
06	DISPLAY BUFFER #6 DISP6
05	DISPLAY BUFFER #5 DISP5
04	DISPLAY BUFFER #4 DISP4
03	DISPLAY BUFFER #3 DISP3
02	DISPLAY BUFFER #2 DISP2
01	DISPLAY BUFFER #1 DISP1
00	DISPLAY BUFFER #0 DISP0

FIG. 6-4 The ia 7301 memory map of the lower memory locations. Only 29 locations are used.

other constants before KBDS was called, changing them back to the 7-segment codes just prior to calling the display routine, this would certainly be a very inefficient use of time and memory. These locations are considered to be dedicated, as are the display registers and the image registers, and should not be used for any other purpose other than the ones to which they have been assigned.

Immediately above the display registers in the memory are located the image registers. These occupy 12 bytes of memory, beginning with the accumulator image, followed by the image of the flags, the B register, C register, etc. We will discuss these in detail later.

Above the display buffers in the memory are located the display registers, one for each digit. In practice, the user loads the appropriate display registers with the hexadecimal numbers that he wishes to display. When CONV is called, the contents of the display registers are converted to the 7-segment coded equivalents and stored in the corresponding display buffers. The contents of the display registers are themselves unmodified.

With the arrangement of the display buffers in mind, let's return to the problem of writing our subroutine. As has already been pointed out the H/L registers will act as a pointer for loading the buffers. They are initially loaded with the address of one of the buffers and then will be incremented or decremented just before the contents of the accumulator is moved into the memory. In this way, the accumulator will be moved to a new location on each loop through the subroutine. It will be necessary to keep track of how many times this is done in order to prevent loading locations other than the display buffers. The operation must take place eight and only eight times. This can be accomplished by setting up another register to keep track of the number of times the program loops through the subroutine. This involves some extra instructions and, as it turns out, the fact that the display buffers are at the very bottom of the memory makes the use of an extra register unnecessary.

If the L register is decremented with a DCR instruction just prior to moving the contents into memory, the last pass will result in H and L containing 0000H. This is correct since the last buffer to be loaded when working from the top down will be display buffer #0 which is in location 0000H. If, after moving accumulator to memory, the zero flag is tested by a conditional jump if not zero, we can exit the subroutine after loading the last buffer.

The entire subroutine is shown in Figs. 6-5 and 6-6. The first instruction, MVI A, BLANK, causes the accumulator to be loaded with the code necessary to blank the display. The prior contents of the H, L, B and C registers are then pushed onto the stack to save them. H and L are then loaded with the address of display register #7 + 1 via an LXI H instruction. This loads the H and L registers with the address of the location just above the last display register. However one of the next instructions is a DCR L which reduces H/L by one so that they then point to the first buffer

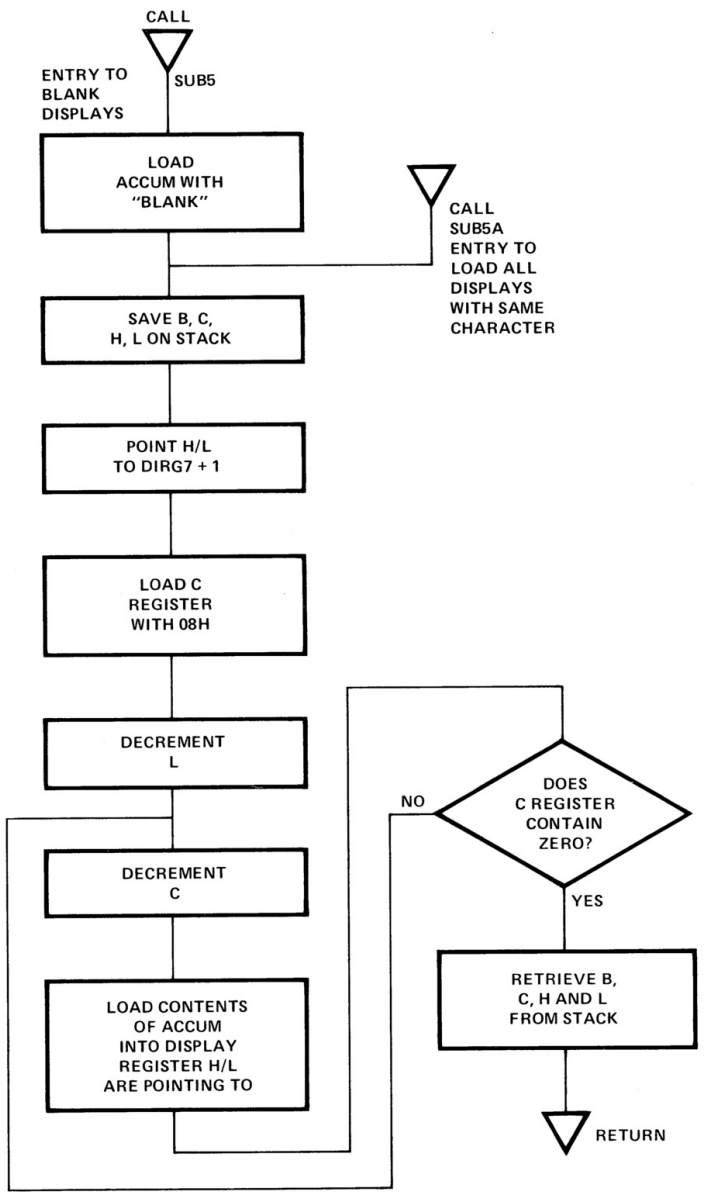

CALL
ENTRY TO BLANK DISPLAYS
SUB5
LOAD ACCUM WITH "BLANK"
CALL SUB5A ENTRY TO LOAD ALL DISPLAYS WITH SAME CHARACTER
SAVE B, C, H, L ON STACK
POINT H/L TO DIRG7 + 1
LOAD C REGISTER WITH 08H
DECREMENT L
DECREMENT C
LOAD CONTENTS OF ACCUM INTO DISPLAY REGISTER H/L ARE POINTING TO
DOES C REGISTER CONTAIN ZERO?
NO
YES
RETRIEVE B, C, H AND L FROM STACK
RETURN

FIG. 6-5 Subroutine 5 is used to load all displays with same character.

to be loaded. The C register is set up as the loop counter by MVI C, 08H. On each pass through the loop, the C register is decremented with a DCR C instruction. The actual loading process is done by the MOV M,A instruction. After the first pass past this point register #7 will contain BLANK. It should be noted that the DCX instruction cannot be used to decrement the H/L register pair since this instruction does not set any of the condition flags.

Since DCR C causes the flags to be set, and the MOV instruction does not affect the flags, the zero flag will still reflect the condition of the C register after the MOV M,A. The JNZ instruction, Jump if Not Zero, causes the program to loop back to the DCR L instruction if C is any nonzero value. This loop will occur seven times, each time loading a new display register with OFFH and decrementing the H/L pair to point to the next register. On the final pass DCR L will result in the zero flag being set, and, after loading the register #0 with BLANK, the JNZ instruction will fail. The program passes through this instruction just as though it were a NOP, No Operation, and goes on to the next instructions. These are a pair of OPO instructions which restore the prior contents of the B/C and H/L pair to their values before calling the subroutine. The final instruction is a RET, Return, which causes program control to return to the main program. The return B, C, H and L contain the same numbers as before the subroutine as do all of the other registers. The only exception is the accumulator which still contains BLANK.

```
80CC 3E11      SUB5A: MVI A, BLANK

80CE E5        SUB5:  PUSH H
80CF C5               PUSH B
80D0 211000           LXI H, DIRG7+1
80D3 0E08             MVI C,8
80D5 2D        SUB5B: DCR L
80D6 0D               DCR C
80D7 77               MØV M,A
80D8 C2D580           JNZ SUB5B
80DB C1               PØP B
80DC E1               PØP H
80DD C9               RET
```

FIG. 6-6 Actual program listing of subroutine 5 shown in the flow diagram of figure 6-5.

Before leaving this subroutine we should point out that a useful option can be added at the very small price of adding a label to the PUSH H instruction. The SUB5A label is needed so the user can call the subroutine. The SUB5B label is needed to provide an address for the beginning of the loop; the JNZ has to reference this label. By assigning a third label, SUB5, to the PUSH H instruction we allow the user to call the subroutine at this point instead of at the MVI A instruction. The effect is that calling SUB5A blanks the displays. Calling SUB5, on the other hand, bypasses the loading of the accumulator and allows the subroutine to procede with whatever was in the accumulator when the routine was called. One useful application is to load the accumulator with the code for a dash. Calling SUB5 will then result in the entire display being loaded with dashes instead of being blanked. This feature is used several times in the ia7301 monitor.

LOADING THE DISPLAY REGISTERS INTO THE ACCUMULATOR

Each of the display registers contains one hexadecimal digit that requires four bits to sepcify. The four bits used are the four LSB of the register; the four MSB are not used at all. There will be many times during the course of the monitor when we will need to move the contents of the display registers into the accumulator. Since the accumulator is an 8-bit register it will be necessary to merge two of the registers together to form one 8-bit word that will fill the accumulator. This happens so often that we have written a special subroutine just for that purpose. It normally is used with display registers 0 and 1 although sometimes other registers are used.

There are several ways this can be done. Our approach is to point H/L at the most significant of the two display registers to be used and to move the contents of that display register into the accumulator. The four MSB are zeroed out and then the contents are shifted four places to the left. Since this is the most significant display register, after the shifting operation the bits representing its contents are in the four MSB positions. Because of our zeroing out the four MSB prior to shifting, these zeroes are shifted around the end of the accumulator and into the four LSB positions. This number is then stored temporarily in the B register. The H/L registers are decremented so they now point to the second, less significant, display register. The contents of this display register are

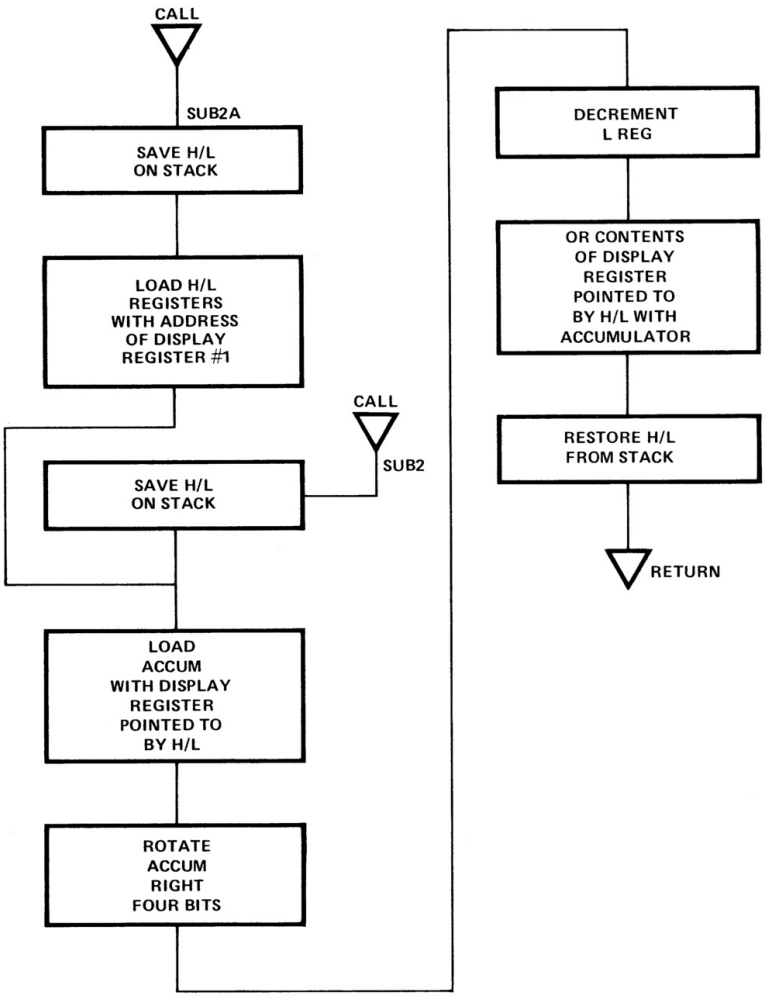


FIG 6-7 Flow diagram of subroutines 2 and 2A used to load the accumulator with the contents of two display registers.

```
808C E5        SUB2A: PUSH H
808D 210900           LXI H, DIRG1
8090 E5        SUB2:  PUSH H
8091 7E               MØV A,M
8092 07               RLC
8093 07               RLC
8094 07               RLC
8095 07               RLC
8096 2D               DCR L
8097 B6               ØRA M
8098 E1               PØP H
8099 C9               RET
```

FIG. 6-8 Program listing of SUB2 and SUB2A.

then moved to the accumulator and the four MSB again zeroed out. ORing the contents of the B register with the accumulator serves to merge the two numbers into the accumulator.

The actual subroutine is illustrated in Figs. 6-7 and 6-8. The H/L register pair is loaded with the address of display register #1 by using an LXI H. MOV A,M then brings the contents of the first display register into the accumulator; four RLC instructions cause the accumulator to be rotated four bit positions to the left. The contents of the display register are now in the four MSB positions. The four LSB positions are occupied by zeroes. The H/L registers are decremented to point at the next display register. Since this is a less significant display than the one already accessed, its contents must become the four LSB of the accumulator. ORA M brings them into the accumulator as the four LSB. The 8-bit number in the accumulator is now the correct combination of the contents of the two display registers. Because we desire to return from this subroutine with the H/L registers pointing to display register #1, a POP H is used to restore the pointer to that value, and a RET instruction causes the return to the calling program.

By calling SUB2A the H/L registers will automatically be loaded with the address of display register #1. An extra label, SUB2, has been added and if the subroutine is called at that point it will procede with the values already present in H and L. This allows the user to use the subroutine with display registers other than 0 and 1.

STORING DISPLAYS IN D AND E

During the operation of the ia7301 the system is frequently called upon to store and retrieve the contents of register pairs in and out of the display registers. Since a register pair holds sixteen bits and each display register stores a four bit hexadecimal number, four display registers are needed to fill the register pair. This problem occurs frequently in the monitor since four of the displays are used to indicate the address of the instruction being accessed, and it becomes necessary to load a register pair with this address. There are actually two operations implied here. One is the loading of the four display registers with the contents of the register pair. The second is the retrieving of the display registers and loading their contents into the register pair; it is this operation that we are presently concerned with. The solution is illustrated in Figs. 6-9 and 6-10.

Addresses are always displayed in dis-

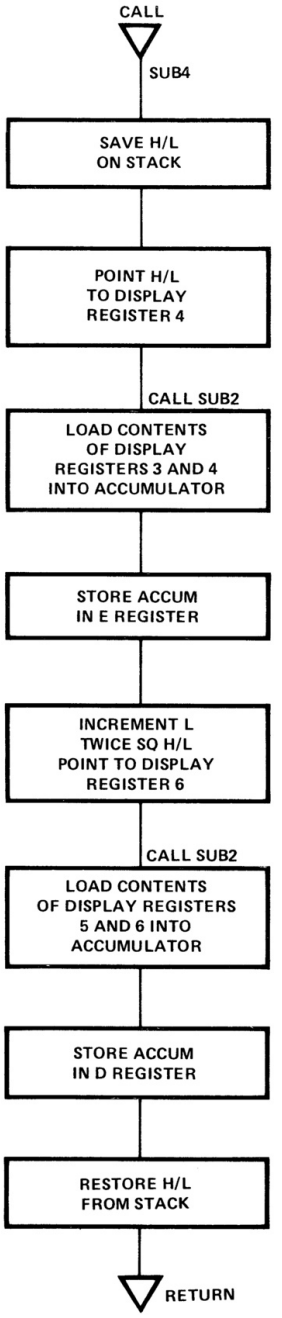


FIG. 6-9 Subroutine 4 is used to load the contents of display registers 6, 5, 4 and 3 into the D and E registers. The program is simplified by calling SUB2 to perform the transfer from display registers to accumulator.

```
80BC E5          SUB4:  PUSH H
80BD 210C00             LXI H, DIRG4
80C0 CD9080             CALL SUB2
80C3 5F                 MØV E,A
80C4 2C                 INR L
80C5 2C                 INR L
80C6 CD9080             CALL SUB2
80C9 57                 MØV D,A
80CA E1                 PØP H
```

FIG. 6-10 Program listing of SUB4.

plays 3, 4, 5, and 6. That leaves display 7 open for a mode symbol and displays 0 and 1 for the contents of the memory location being accessed by the address on the left. Display 2 carries a dash to separate the address from the instruction. The fact that displays 3, 4, 5 and 6 are the only ones used to display addresses simplifies our problem since we can load H and L with these addresses without having to make provisions for the user deciding to use other displays.

The key to the subroutine is in realizing that the best approach is to handle two displays at a time. This now becomes very similar to the subroutine we just discussed—that of loading the accumulator with the contents of two display registers. That subroutine took 14 bytes of memory to code and, if we are clever, we can use it over here. We'll point H/L to the most significant of one of the display register pairs, say #4 of the 3 and 4 combination. Before doing that a PUSH H will save the prior contents of the H/L registers for the user. We can then call SUB2 which will cause the accumulator to be loaded with the contents of the display registers 3 and 4. Move these to register E with a MOV E,A instruction.

We have now succeeded in loading displays 3 and 4 into the E register. If we now increment the L register with two INR L instructions it will point to display register #6. Calling SUB2 a second time will load the accumulator with the contents of registers 5 and 6. A MOV D,A loads this data into register D. All that remains is to POP H to restore the H and L registers to their original values and the subroutine is completed. This is an example of how one subroutine can be used to build another. If the problems are carefully defined at the beginning and are small enough, the program will become one of subroutines building upon other subroutines.

THE SHIFT AND ENTER SUBROUTINE

It's time to consider a subroutine that is a little more involved. During the KBDS subroutine, the situation often occurs where the system must take the data keyed into the keyboard by the user and enter it into the displays. Each time a key is depressed it must appear on the right-most display. The prior contents of the displays are each shifted one digit to the left. We define the display field by the number of digits that will appear, usually either two or four, and by the left-most display that will be used to display data. When addresses are being entered on the keyboard, they will appear in displays 3, 4, 5 and 6; the field will be four digits

wide. When registers or memory locations are being modified, they will appear in displays 0 and 1; in this case the field is two digits wide. One exception is for modifying the stack pointer or the program counter. These registers both are 16 bits long and use displays 0, 1, 2 and 3.

The actual subroutine is shown in Figs. 6-11 and 6-12. The reader should refer to this frequently during the following discussion. The subroutine is used by storing the number of digits in the field in register C and pointing H and L to the most significant display register that will be included in the field. We will provide the user with two options that automatically load this data into the appropriate registers for two of the most common situations. A third option allows the user to load the registers himself.

In any event the H, L and C registers will be used during the subroutine to store data. To avoid destroying the prior contents of these registers they are PUSHed onto the stack with a PUSH H and a PUSH B instruction. The H and L registers are then loaded with an LXI H, DIRG1 and the C register with a MVI C, 02H. This points H/L to display register #1 and sets the field size at 2. JMP SUB1D then takes the program to the next step.

```
8068 E5        SUB1A: PUSH H
8069 C5               PUSH B
806A 210900           LXI H, DIRG1
806D 0E02             MVI C, 02H
806F C37E80           JMP SUB1D
8072 E5        SUB1B: PUSH H
8073 C5               PUSH B
8074 210E00           LXI H, DIRG6
8077 0E04             MVI C, 04H
8079 C37E80           JMP SUB1D
807C E5        SUB1C: PUSH H
807D C5                PUSH B
807E 0D        SUB1D: DCR C
807F 2D               DCR L
8080 46               MØV B,M
8081 28                INR L
8082 70               MØV M,B
8083 2D               DCR L
8084 0D               DCR C
8085 C27F80           JNZ SUB1D+1
8088 77               MØV M,A
8089 C1               PØP B
808A E1               PØP H
808B C9               RET
```

FIG. 6-12 Program listing for SUB1A, B and C.

The JMP instruction is necessary because we want to give the user some options. If he calls SUB1B the same process occurs except H/L are pointed to display register #6 and the C register is loaded with the field size of four. If the JMP SUB1D was not included after the first set initializing conditions the computer would go on to the second set, those represented by SUB1B, and execute those. This would cause the first set of conditions to be erased. A second JMP SUB1D provides the second set of initializing conditions the same jump ahead as the first.

The third option, SUB1C, allows the user to enter the subroutine after loading his own data into H/L and C. The fact that the first two options both begin by pushing B and H means that the subroutine must end by two corresponding POPs to restore these registers. Thus at the end of the subroutine the stack pointer will be incremented by four (due to the two POP instructions) regardless of which option the user used to enter the subroutine. This applies also to option SUB1C. If the third option is used, the values already present in H/L and C when the subroutine was called will be used. However, at the conclusion of the subroutine the stack pointer will be incremented by four even though the stack was not used for saving H/L and C as in the first two options. This is unacceptable since it means the stack will steadily decrease every time option C is used.

It is very important that the stack size stay constant while entering or exiting a subroutine. Another way to look at this is to say that for every PUSH there must be a corresponding POP. But if the user enters via option C there will be two POPs at the conclusion of the subroutine but no PUSHes at the beginning. This is corrected by inserting a PUSH H and a PUSH B to keep the stack size constant in the event that the subroutine is entered via this option. The fact that H is pushed means nothing in terms of register H; it remains intact through the execution of the instruction which only causes its contents to be duplicated into the memory location addressed by the stack pointer. The PUSH does however decrement the stack pointer by two which balances out one of the POPs at the end of the subroutine. A second dummy push balances the second POP. When used in this way it is academic which register is PUSHED, since the reason was to decrement the stack pointer and not to save the register itself.

The subroutine itself really begins at

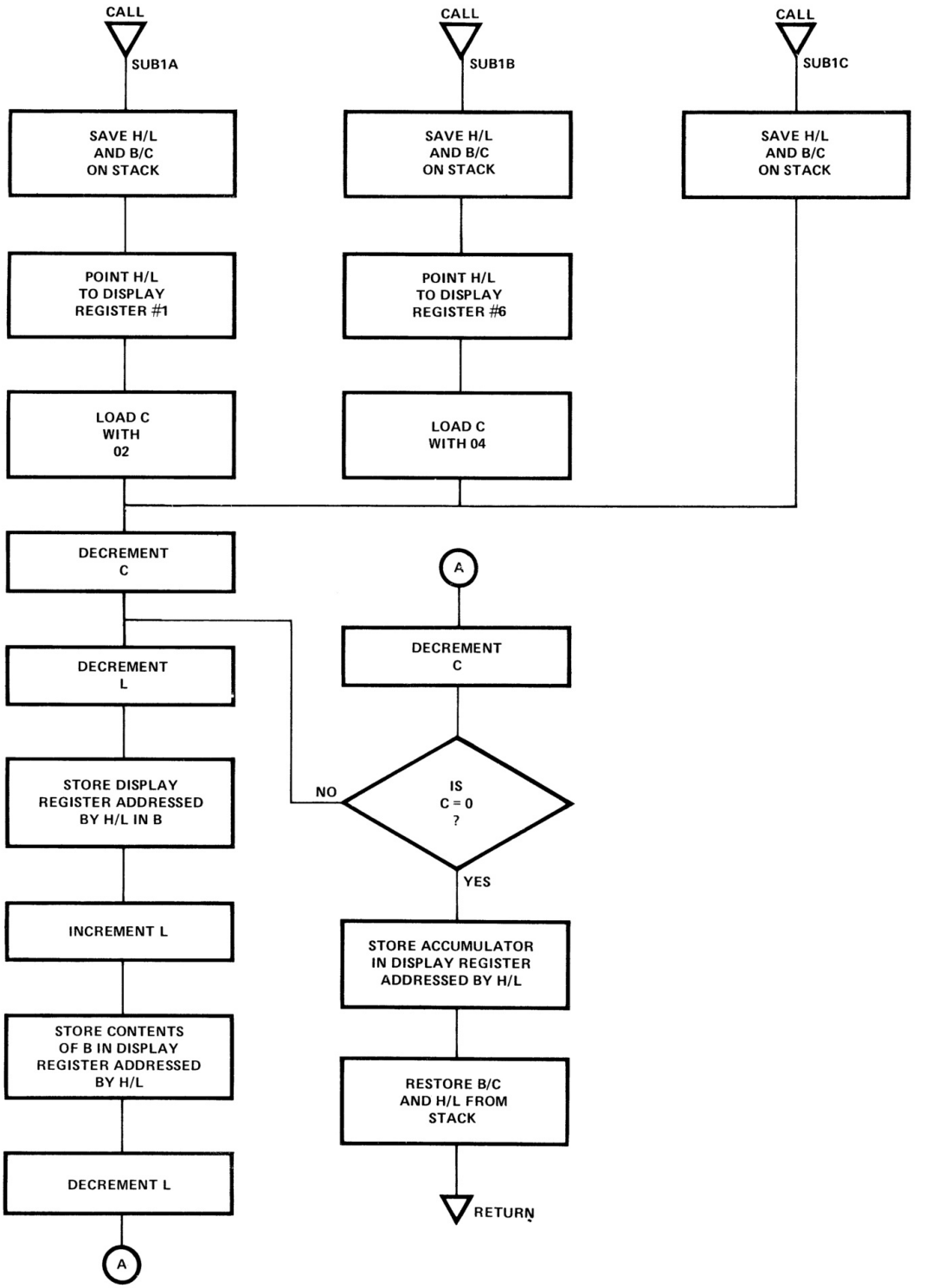


FIG. 6-11 Flow diagram for SUB1A, B and C used to enter and shift data into the display registers. The subroutines are entered with the new byte of data in the accumulator.

SUB1D with a decrement register C. C, in this case, acts as the loop counter. When it reaches zero, we will have gone through the subroutine once for each digit in the display. We must then point H/L to the adjacent, less significant display. This is done so that we can move the data in that display register to the adjacent, more significant display register. This is the shifting operation that occurs in the left-hand direction. Of course, we cannot simply move data from one memory location to another; that instruction does not exist in the 8080 instruction set. We can, however, move it first to a temporary holding register and then from there to the other memory location. The B register is used for this purpose. The MOV B,M stores the data in the holding register, L is then decremented to point to the more significant display register, and a MOV M,B transfers the data into it.

But what about the data that was in the more significant display register to begin with? Won't that be lost? The answer is yes; the data in the most significant display register at the edge of the display field is lost when new data is keyed in at the right. This allows the user to correct erroneous data by simply cleansing the display by shifting in completely new data at the right. The next time through the loop the shifting action will be taking place further down in the display. The new data is shifted into one of the registers, but the previous contents of the new register are not lost since they were shifted to a third register on the previous pass through the loop. After the shifting operation is completed the L register is incremented to point it back to the more significant of the display register pair and the C register decremented to keep track of the passes through the loop. A JNZ SUB1D tests to see if the C register is at zero yet. A zero here means that the last pass through the loop shifted data out of display register #0. If not, the jump to SUB1D occurs and the loop is repeated with another display register pair.

If the zero test passes, meaning the C register contains zero and the display register #0 has had its contents shifted to register #1, the new data must be shifted into the register #0. This data is presently being stored in the accumulator. This subroutine is called when the keyboard display routine finds a valid key closure. That subroutine is exited by storing the hexadecimal code corresponding to the key in the accumulator. The subroutine we are working on is called at that time; the last operation of that subroutine must be to take the new number contained in the accumulator and store it in register #0 to complete the shifting operation. This is done by a MOV M,A instruction. B and H are then POPed off the stack to restore those registers. A RETurn causes the program to return to the main program.

LOADING THE ACCUMULATOR INTO TWO DISPLAY REGISTERS

A very similar problem to the one just discussed is that of loading the contents of the accumulator into two adjacent display registers. Before, a four bit number that was stored in the accumulator had to be transferred into one of the displays while all the rest were shifted one digit to the left. Now we are considering the problem of how to split an 8-bit number in the accumulator into two four-bit numbers and store them in adjacent displays.

The subroutine is illustrated in Figs. 6-13 and 6-14. As before one option automatically loads H/L and C with initializing data. A second option allows the subroutine to be called and run using the prior contents of these registers. We begin by PUSHing H, B and PSW to save these registers. H/L is then loaded with the address of display register #1 with an LXI H, DIRG1 and C is loaded with the field size via an MVI C,02H. We then insert a JMP SUB3C to hop over the other set of initializing conditions for option #2. The second option allows the user to call the subroutine and run using the present contents of the CPU registers. As before we must fake the stack with a PUSH PSW, PUSH H and a PUSH B to balance the POPs that occur at the end of the subroutine.

The main portion of the subroutine begins at SUB3C. During this portion of the program we must modify the contents of the accumulator but must be able to return to the original contents later. We could do this by a MOV B,A and later retrieve them with MOV A,B. This is a perfectly valid method and since the original PUSH B that freed up register C also did the same for register B, this method will not interfere with data already present in B. We chose another method, however. We merely push the accumulator and flags onto the stack memory with a second PUSH PSW. When we need to retrieve it later it can simply be POPed back off the stack. PUSH PSW duplicates the accumulator and flags into the memory. The accumulator and flags are themselves unchanged by this operation and are ready to be modified as required.

The number in the accumulator,

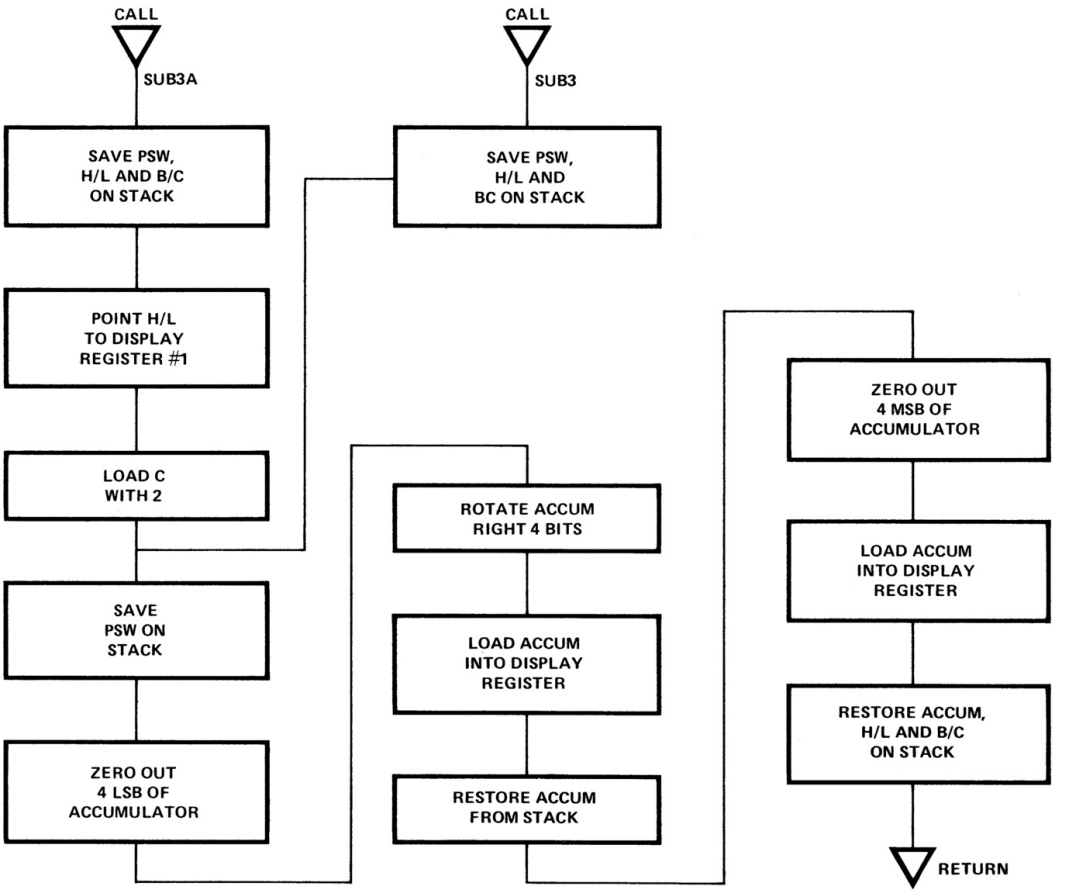

CALL
SUB3A
CALL
SUB3
SAVE PSW, H/L AND B/C ON STACK
SAVE PSW, H/L AND BC ON STACK
POINT H/L TO DISPLAY REGISTER #1
LOAD C WITH 2
SAVE PSW ON STACK
ZERO OUT 4 LSB OF ACCUMULATOR
ROTATE ACCUM RIGHT 4 BITS
LOAD ACCUM INTO DISPLAY REGISTER
RESTORE ACCUM FROM STACK
ZERO OUT 4 MSB OF ACCUMULATOR
LOAD ACCUM INTO DISPLAY REGISTER
RESTORE ACCUM, H/L AND B/C ON STACK
RETURN

FIG. 6-13 Flow diagram of SUB3 and SUB3A. It is used to load the contents of the accumulator into two display registers. If SUB3A is called, display registers #0 and #1 will be used. If SUB3 is called the display registers being pointed to by H/L will be used.

```
809A F5        SUB3A: PUSH PSW
809B E5               PUSH H
809C C5               PUSH B
809D 210900           LXI H, DIRG1
80A0 0E02             MVI C, 02H
80A2 C3A880           JMP SUB3C
80A5 F5        SUB3:  PUSH PSW
80A6 E5               PUSH H
80A7 C5               PUSH B

80A8 F5        SUB3C: PUSH PSW
80A9 E6F0             ANI 0F0H
80AB 0F               RRC
80AC 0F               RRC
80AD 0F               RRC
80AE 0F               RRC
80AF CD7C80           CALL SUB1C
80B2 F1               P0P PSW
80B3 E60F             ANI 0FH
80B5 CD7C80           CALL SUB1C
80B8 C1               P0P B
80B9 E1               P0P H
80BA F1                P0P PSW
80BB C9               RET
```

FIG. 6-14 Program listing for SUB3 and SUB3A.

whether it is data or an instruction, is an eight-bit quantity. The display registers are set up to only handle numbers in the four LSB positions making it necessary to blank out half of the accumulator. This is done with an ANI 0F0H which zeroes out the four LSB. The remaining four MSB are then shifted into the now vacant four LSB positions by four RRC instructions. SUB1C is now called and since we have preconditioned H/L and C, this causes the accumulator contents to appear in display register #0. Notice that the number that apppears is actually the most significant half of the original contents of the accumulator before the blanking and shifting process.

Now for the second half. POP PSW to restore the accumulator to the original number. Blank the most significant half with an ANI 0FH and then CALL SUB1C. The subroutine then moves the contents of display register #0 into display register #1 and the new contents of the accumulator, which now contains only the least significant half of the number, into display register #0. It only remains to POP B, POP H and POP PSW to restore the registers and RETurn.

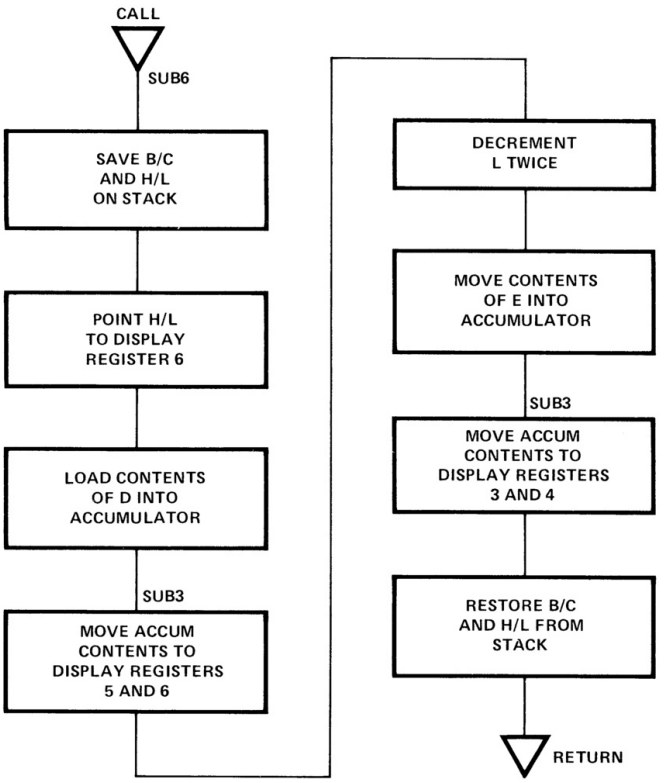

CALL
SUB6
SAVE B/C AND H/L ON STACK
POINT H/L TO DISPLAY REGISTER 6
LOAD CONTENTS OF D INTO ACCUMULATOR
SUB3
MOVE ACCUM CONTENTS TO DISPLAY REGISTERS 5 AND 6
DECREMENT L TWICE
MOVE CONTENTS OF E INTO ACCUMULATOR
SUB3
MOVE ACCUM CONTENTS TO DISPLAY REGISTERS 3 AND 4
RESTORE B/C AND H/L FROM STACK
RETURN

FIG. 6-15 Flow diagram of SUB6 used for loading the contents of the D and E registers into display registers 3, 4, 5 and 6.

```
80DE E5          SUB6:  PUSH H
80DF C5                 PUSH B
80E0 210E00             LXI H, DIRG6
80E3 7A                 MØV A,D
80E4 0E02               MVI C,02H
80E6 CDA580             CALL SUB3
80E9 2D                 DCR L
80EA 2D                 DCR L
80EB 7B                 MØV A,E
80EC CDA580             CALL SUB3
80EF C1                 PØP B
80F0 E1                 PØP H
80F1 C9                 RET
```

FIG. 6-16 Program listing for SUB6.

Before leaving this subroutine we should point out that there is another way to decrement the stack pointer without using PUSH instructions. The instruction DCX, Decrement Register Pair, is just as valid when used with the stack pointer. Thus two DCX SP instructions can be used in place of a PUSH instruction for option SUB3A.

LOADING THE CONTENTS OF A REGISTER PAIR INTO DISPLAYS 3, 4, 5, AND 6

We will consider one more subroutine before going on to more complex problems. We have already considered the question of loading the contents of display registers 3, 4, 5 and 6 into the D and E registers. Just as important is the problem of loading the D and E registers back into the same display registers. The subroutine that accomplishes this is illustrated in Figs. 6-15 and 6-16.

Since this subroutine makes use of the H/L and C registers we will PUSH H and B to save the prior contents of these registers. H/L is then preconditioned by loading it with the address of the most significant display register, DIRG6, and the C register with 02 to define the field size. Even though the total field is four digits wide we're going to treat it in two digit nibbles so we can use SUB3 that we just wrote. Before doing that we must move the contents of the D register into the accumulator with a MOV A,D instruction. All of the registers are now in the proper condition to CALL SUB3. That subroutine causes the contents of the accumulator, formerly the contents of D, to be loaded into display registers 5 and 6. Register L is then decremented twice with two DCR L instructions to point H/L to display register 4. Register E is then MOVed into the accumulator and SUB3 CALLed a second time to display it in display registers 3 and 4. B and H are POPed to restore them to their prior contents and a RET executed to return program control to the calling program.

This completes the set of preliminary subroutines needed for the main program segments. In addition to those just discussed three others of a more complex nature will be needed. KBDS will display the contents of the display registers and find valid key closures. MDCCK determines if the image of the key closure contained in the accumulator when the exit occurs is one of the hexadecimal keys or is a mode key. If the key closure is a mode key MDCCK is exited with a JUMP to the program segment that services that mode of operation. If, on the other hand, the key closure is a hexadecimal key, meaning data, the subroutine

is exited via a return to the calling program. The third subroutine, CONV, causes the contents of the display registers to be converted into seven-segment codes which are loaded into the display buffers. These subroutines are normally used in the order CALL CONV, CALL KBDS, CALL MDDCK. By calling CONV first, the new contents of the display registers are converted to seven-segment code prior to the keyboard display routine so the most current data will be displayed. It would be possible to combine all three of the subroutines so that one CALL would execute all three. By leaving them separate however the user is provided with more flexibility. If, for instance, he were writing a program for using the ia7301 as a super calculator he would need some keys for the arithmetic operations he wished to perform. As it is, each time one of the mode keys is depressed a JUMP occurs to that program segment. But by writing his program so that CALL CONV and CALL KBDS occur but the MDCCK subroutine is omitted, he is free to dedicate the mode keys to any function he wishes.

At this point in our discussion we are going to leave KBDS, CONV and MDCCK. Since we are trying to make this chapter follow as closely as possible the actual practice of writing the system monitor we intend to do the same jumps as we did at that time. In fact, the subroutines, SUB1 through SUB6, were written first. Then the DCM program segment was written, and during the process of writing that segment the definitions of KBDS, CONV and MDDCK were arrived at. The DCM program was begun with only a vague idea of the nature of these three subroutines. By the time DCM was completed they had been completely defined and were then written.

DCM – THE DISPLAY AND CHANGE MEMORY MODE

When the system is first energized, or when the reset button is depressed, the monitor is entered at location 8000H. Some initializing instructions are executed (these will be covered later) and the DCM mode is entered. Because the DCM program follows the initializing instructions with no intervening program segments, no JUMP to the DCM program will be necessary.

The first operation of the DCM program must be to load all of the displays with dashes except for display register #7 which is to be loaded with the DCM mode symbol referred to here as DECEM. This can be done by loading the accumulator with the DASH symbol, MVI A, DASH and then calling SUB5 to load this symbol into all of the displays. H/L are then pointed to display register #7 by LXI H, DIRG7. MVI M, DECEM loads the DCM mode symbol into this display register. This initial portion of the program is illustrated in Fig. 6-17.

When the DCM mode is first entered the user sees only dashes and the DCM symbol in display #7. The system then idles waiting for the user to enter a starting address into the keyboard. To do this we CALL CONV1, the starting address of the subroutine that converts the contents of the display registers into seven segment code. When this subroutine is completed the display buffers will contain the code of the numbers to be displayed; a return is executed to the next instruction of DCM. This is a call to KBDS0 which causes the contents of the display buffers to appear in the LED displays. At the same time the displays are activated the subroutine searches the keyboard for a key closure. We will discuss the workings of KBDS0 shortly; for now it is enough just to know that when this subroutine is completed the image of the key closure will be in the accumulator. Finally MDCCK is called to determine if the key closure represents a hexadecimal key or one of the mode keys. If the closure was a mode key the subroutine is exited via a JUMP to the correct mode program. If the key is a hexadecimal key, the subroutine is exited via a RETurn to the calling program. The key image is still in the accumulator.

MDCCK checks for closures for all but one of the mode keys. The exception is the NEXT key. It is not considered during MDCCK because NEXT can be used in several of the modes. There is no way the system can tell during the MDCCK subroutine which of the modes it was in when NEXT was found. This means that NEXT must be checked after returning from MDCCK; the system will react to this key in different ways depending on which mode it is in. Thus, the return from MDCCK is made with either a hexadecimal number in the accumulator or the image of the NEXT key.

The instruction following CALL MDCCK presumes that either a number or the NEXT image is in the accumulator. If it is not, there must have been a mode change and the program execution will not be at this point but will be from one of the mode programs. Our first task is to determine whether we have a number or NEXT. This is done with a CPI NEXT. If the NEXT key was depressed the result of the CPI will be the setting of

the zero flag. This can be tested by a conditional jump, JZ DCM3. If this test fails, we have a number. CALL SUB1B to shift this number into the address field of the display. JMP DCM2 causes the program to loop back to CALL CONV1 and the process begins again. The effect to the user is that he energizes the system, the display lights up with dashes except for the DCM symbol in the far left display. He presses a hex key to load the most significant hex digit of the address he wishes to examine. That number immediately appears in display #3 which is the least significant digit of the address field. The system is now back in KBDS1 looping; the displays are enabled and the keyboard is continually being scanned waiting for another key closure.

When the user depresses another hex key, the process begins again. Now when SUB1B is called the digit already displayed is shifted one place to the left and the new key entry appears in the right-hand display of the address field. This occurs each time a key is depressed. When the numbers being keyed in have been shifted so that the address field is filled, additional key entries will result in the new key appearing at the right while the left-most digit is lost. This allows the user to correct errors by shifting in the correct address and simply letting it displace the incorrect one. The entire DCM program is illustrated in a flow diagram in Fig. 6-18.

The DCM2 loop can be exited in two ways. First, the MDCCK subroutine may show that the user decided to switch to another mode of operation. In this case the loop will be exited via a JUMP to that mode program directly out of MDCCK. Or, what is more likely, he will depress the NEXT key when he is satisfied with the address that has been entered and is now being displayed. This has the effect of causing the CPI NEXT test to set the zero bit and the JZ DCM3 to occur. The DCM3 portion of the program appears in Fig. 6-19.

When DCM3 is reached it can only be because of the NEXT key being depressed. We must now do several things. First, the user is evidently satisfied with the address that he sees being displayed. We must therefore access this location and retrieve the contents of that memory location to display them in displays 0 and 1. CALL SUB4 causes the contents of display registers 3, 4, 5 and 6 to be loaded into CPU registers D and E. We can then use these registers as a pointer by an LDAX D instruction which causes the contents of the memory location addressed by D/E to be loaded into the accumulator. CALL SUB3A causes the contents of the accumulator to be loaded into display registers 0 and 1, the contents portion of the display. Of course, these numbers will not actually appear until CONV1 and KBDS1 are called. Once again we also must call MDCCK to be sure that the key closure is not one of the mode keys. Assuming it is not, the system will execute the next instruction in the DCM program, CPI NEXT, to set the zero bit if the NEXT key is depressed. If so the JZ DCM5 will occur. If the key was a hex numeric key, the CPI and JZ instructions have no effect. CALL SUB1A causes the hex key to be loaded into display register #0. JMP DCM4 sets up a loop back to CALL CONV1. We now have the same situation as before. Now each time a new hex key is depressed it appears in display #0. The digit that was in #0 is shifted to display #1 and the former contents of display #1 are lost. The system loops on this program segment until a mode key is depressed or the CPI and JZ test finds a NEXT key depression. DCM5 is illustrated in Fig. 6-20.

When the NEXT key is depressed now it means the user is satisfied with the contents of the memory location being displayed. Of course, he may have been satisfied when his location was accessed and the present contents displayed. If so, he depressed NEXT and went on. If he wished to change the contents, he did so by keying in the new value. In any event, NEXT is our clue to go on. Before doing that, however, we have to take the new contents as modified by the user and load them back into the location being displayed. Rather than worry about whether or not they have been changed, we will simply load them into memory regardless. This is the basic function of DCM5. CALL SUB4 fetches the contents of display registers 3, 4, 5 and 6 and merges them into the D and E registers. SUB2A

```
8021 3E10      DCM1:  MVI A, DASH
8023 CDCE80           CALL SUB5
8026 210F00           LXI H, DIRG7
8029 3612             MVI M, DECEM
802B CD3281    DCM2:  CALL CØNV1
802E CDF280           CALL KBDSO
8031 CD4E81           CALL MDCCK
8034 FE12             CPI NEXT
8036 CA3F80           JZ DCM3
8039 CD7280           CALL SUB1B
803C C32B80           JMP DCM2
```

FIG. 6-17 Program listing of the initial portion of the DCM mode routine.

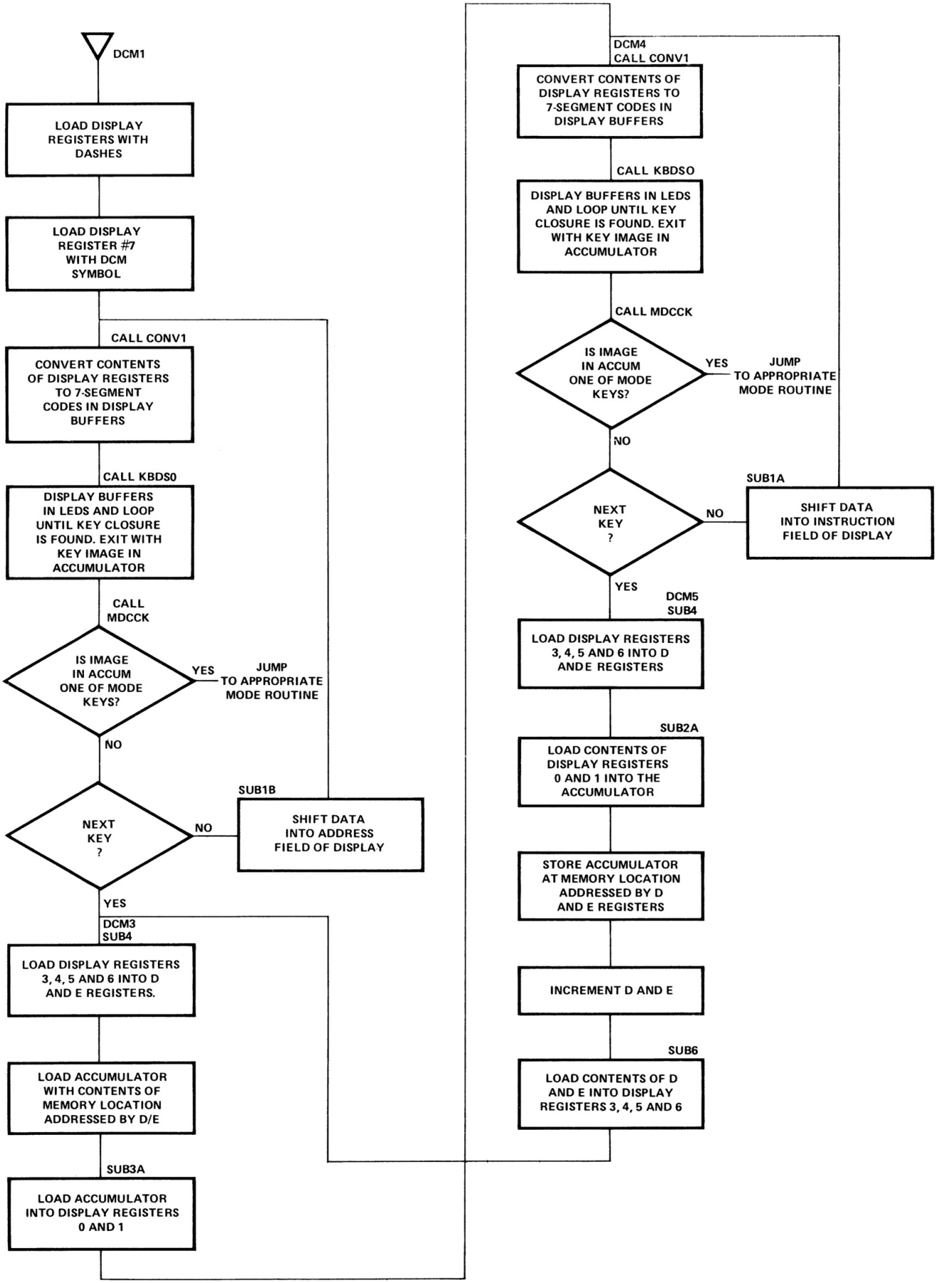

DCM1
LOAD DISPLAY REGISTERS WITH DASHES
LOAD DISPLAY REGISTER #7 WITH DCM SYMBOL
CALL CONV1
CONVERT CONTENTS OF DISPLAY REGISTERS TO 7-SEGMENT CODES IN DISPLAY BUFFERS
CALL KBDS0
DISPLAY BUFFERS IN LEDS AND LOOP UNTIL KEY CLOSURE IS FOUND. EXIT WITH KEY IMAGE IN ACCUMULATOR
CALL MDCCK
IS IMAGE IN ACCUM ONE OF MODE KEYS?
YES
JUMP TO APPROPRIATE MODE ROUTINE
NO
NEXT KEY ?
NO
SUB1B
SHIFT DATA INTO ADDRESS FIELD OF DISPLAY
YES
DCM3
SUB4
LOAD DISPLAY REGISTERS 3, 4, 5 AND 6 INTO D AND E REGISTERS.
LOAD ACCUMULATOR WITH CONTENTS OF MEMORY LOCATION ADDRESSED BY D/E
SUB3A
LOAD ACCUMULATOR INTO DISPLAY REGISTERS 0 AND 1
DCM4
CALL CONV1
CONVERT CONTENTS OF DISPLAY REGISTERS TO 7-SEGMENT CODES IN DISPLAY BUFFERS
CALL KBDSO
DISPLAY BUFFERS IN LEDS AND LOOP UNTIL KEY CLOSURE IS FOUND. EXIT WITH KEY IMAGE IN ACCUMULATOR
CALL MDCCK
IS IMAGE IN ACCUM ONE OF MODE KEYS?
YES
JUMP TO APPROPRIATE MODE ROUTINE
NO
NEXT KEY ?
NO
SUB1A
SHIFT DATA INTO INSTRUCTION FIELD OF DISPLAY
YES
DCM5
SUB4
LOAD DISPLAY REGISTERS 3, 4, 5 AND 6 INTO D ANDE REGISTERS
SUB2A
LOAD CONTENTS OF DISPLAY REGISTERS 0 AND 1 INTO THE ACCUMULATOR
STORE ACCUMULATOR AT MEMORY LOCATION ADDRESSED BY D AND E REGISTERS
INCREMENT D AND E
SUB6
LOAD CONTENTS OF D AND E INTO DISPLAY REGISTERS 3, 4, 5 AND 6

FIG. 6-18 Flow diagram of the DCM program.

```
803F CDBC80    DCM3:   CALL SUB4
8042 1A                LDAX D
8043 CD9A80            CALL SUB3A
8046 CD3281    DCM4:   CALL CØNV1
8049 CDF280            CALL KBDSO
804C CD4E81            CALL MDCCK
804F FE12              CPI NEXT
8051 CA5A80            JZ DCM5
8054 CD6880            CALL SUB1A
8057 C34680            JMP DCM4
```

FIG. 6-19 Program listing of DCM3 and DCM4.

```
805A CDBC80    DCM5:   CALL SUB4
805D CD8C80            CALL SUB2A
8060 12                STAX D
8061 13                INX D
8062 CDDE80            CALL SUB6
8065 C33F80            JMP DCM3
```

FIG. 6-20 Program listing of DCM5.

then causes the contents of display registers 0 and 1 to be loaded into the accumulator. STAX D causes the contents of the accumulator, which now contains the modified contents of the memory location, to be stored in the location addressed by D/E. Increment the D/E register pair and call SUB6 to load the new value of D/E back into display registers 3,4,5 and 6. A jump to DCM3 completes the subroutine. This last instruction causes a loop back to the point in the program where the contents of the display registers are fetched to D and E and used to access the memory.

The DCM program segment illustrates the way in which our short subroutines can be put together to build a major program segment. DCM is one of the most important aspects of our monitor. It can be used to examine and modify any of the memory locations which in turn enables the user to write programs. He merely initializes the DCM address at the starting address of his program and enters the first instruction. Each time the NEXT key is depressed, another instruction can be entered.

THE KEYBOARD DISPLAY SUBROUTINE

By the time the DCM program segment was completed the requirements of KBDS, CONV and MDCCK were clear. The first to be tackled was KBDS.

The first problem is to save the CPU registers. This is done by PUSHing B, D and H. PSW is skipped because the goal of the KBDS subroutine is to return to the calling program with the image of a key closure in the accumulator. Thus there is no point in saving the accumulator; it will be used to carry data out of the subroutine. Once the registers have been saved, the CPU must be initialized.

The problem of the keyboard scans is complicated greatly by the fact that the keys may bounce when being closed or opened. Of course, more expensive keys could be used, but this is a perfect opportunity for the microprocessor to reduce the cost of the system by programming out the bounces. We do this by initializing the C register as the bounce counter. A number, BONCE, is chosen large enough to be sure that when the keyboard has been scanned this many times any bounces will have died out. MVI C, BONCE sets up the bounce counter. The D/E registers are pointed to row 7 of the keyboard by MVI D, KER07. It is not necessary to use an LXI instruction in this case, since, as we saw in the last chapter, only the high order eight address lines are used to address the

keyboard rows. By using a MVI instruction instead of an LXI instruction, one byte instead of two bytes, a byte of memory is saved. Finally, the H/L registers are pointed to display buffer #7 with an LXI instruction. See Figs. 6-19 and 6-20.

Having set up the register pairs and the bounce counter, we begin the actual program by moving the contents of the display buffer into the display. This is a two part process which begins with a MOV A,M which brings the contents of display buffer #0 into the accumulator. STAX D then sends it out on the data bus. Because any memory write operation to the block, as this is, results in the display latches picking up the data, the STAX D operation causes the contents of the accumulator to be displayed. The number will not be displayed in all of the displays of course, but only in the 7-segment display that is being enabled. In the first pass through the program, this will be display #7. At the same time the address in the H/L registers is that of display buffer #7 so that the number displayed will appear in the correct digit position within the display.

Once the numeric data has been sent to the display latches that data will remain in the latches until it is replaced. We go on to the next instruction, LDAX D, which reads the internal input port into the accumulator. Again, any memory read operation done from the CXXX memory block will result in the reading of the internal input port into the accumulator. The three LSB of this port are the keyboard columns. Bit position #6 is the output of the one-shot that insures none of the displays is on too long. The one-shot is equipped with an RC combination to produce a timing output pulse of about 1 ms. This output pulse is sent to the master reset pin of the display latches; when the one-shot times out the display is blanked so that the high LED currents are turned off preventing damage to the display. One millisecond is a long time in the world of the microprocessor and we will want to be sure that this period has elapsed before going on with the program. Otherwise the 8080 may race on ahead to the next digit thus shutting off the first display prematurely. For this reason, the output of the one-shot is also brought to the multiplexer that implements the internal input port.

The monitoring process is done by an ANI 40H which zeroes out all of the accumulator bits except for bit #6. We then test this bit for non-zero contents by a JNZ KBDS3+1. Thus the JNZ, Jump if Not Zero, causes the system to loop around KBDS3+1 until the one-shot times out. When this occurs, the 8080 will detect a zero on the bus, the JNZ test will fail and the system will go on to execute the next instruction. This is a second LDAX D which serves to reload the accumulator with the data on the internal input port. Once again, certain of the accumulator bits are zeroed out. This time an ANI 0FH allows only the four LSB to remain unchanged. These are the three keyboard column inputs in bit positions 0, 1, and 2 and the output of the step one-shot in bit position #3. To test for a key closure here we execute a JNZ KBDS5. Thus if either the step key or the mode or hexadecimal number keys are depressed the program execution will continue at KBDS5. See Fig. 6-20.

At this point we come to a branch in the program, see Fig. 6-19. The problem is to separate the key bounces from the real key closures. Otherwise, the system runs so fast that it may process data from the first key closure and be back scanning the keyboard in time to pick up a bounce from the original key closure. Naturally we won't want this bounce to be interpreted as a second, different key closure. The program used in the ia7301 monitor requires that the user not press any keys until BONCE time has elapsed after entering the KBDS subroutine. Then the monitor will wait a BONCE time again after finding a key closure before processing the closure. This gives any bounces a chance to die out before the key is processed; the key is checked frequently during this process to be sure the closure is not a bounce.

The basic decision is made by the jump to KBDS5. Either a step key or a mode or number key will cause a jump to this point. Assume for a moment that the tests fail, that is there are no key closures to process. The system then goes on past the jump instructions to DCR C which decrements the contents of the bounce counter originally set at BONCE. The DCR instruction sets the various CPU flags so that the bounce counter can now be easily checked to see if it is zero. Naturally on this first pass it will contain BONCE-1, so the test, JNZ KBDS4, causes the program to go on to KBDS4.

KBDS4 is used to decrement D and L. This has the effect of moving the D/E pointer to row #6 and the H/L pointer to display buffer #6. After decrementing the two pointers a JMP KBDS2 causes the program to loop back to KBDS2 which causes the contents of the display buffer to be MOVed to the accumulator and displayed. If there are no key closures the

program will scan down the display buffers, sending each to the display latches and enabling the correct digit display through the D/E pointer. There is one more jump instruction required, a JM KBDS1, just prior to the JMP KBDS2. This is for the case when the system has already reached the display buffer #0. Now when KBDS4 is reached and L is decremented, the L register will contain −1, a negative number. The JM test recognizes this and correctly causes a jump back one step earlier in the initialization program to the point where D/E and H/L are first loaded. This provision is added to make the system begin a fresh scan at row #7 again. The bounce counter, register C, is unaffected by this.

Each time a row of keys is tested the bounce counter is decremented so that as time goes on its contents are steadily decreasing. At some point it will contain zero and the JNZ KBDS4 test will fail. When that happens the system passes over this instruction to INR C and then on to the next instruction which is, in fact, the first instruction of KBDS4. This confusing sequence of events can be seen clearly in the flow diagram. Each pass through the program reduces the bounce counter by one until it finally reaches zero. Thereafter each pass decrements the counter by one, as before, but then increments the counter just before jumping back to the initialization part of the program. The bounce counter alternates between 0 and +1; at the time the loop back occurs, however, it is always +1. The reason for this will become clear shortly.

At some point a key closure will be detected and the KBDS5 portion of the program will be entered. The first instruction of KBDS5 is a DCR C followed immediately by a JZ KBDS6. The jump instruction serves to test the bounce counter to see if it is zero. If it is, the jump occurs to KBDS6 and the key closure is processed. If not, the C register is reloaded with BONCE and a jump to KBDS4 executed. In this mode the system loops as before, decrementing the bounce counter on each pass and testing for zero as before. When the counter reaches zero the JZ KBDS6 causes a jump to KBDS6 which processes the key closure. If the counter does not contain zero it is reloaded with BONCE and a jump to KBDS4 made where the two pointers are decremented as before. The effect is that a key closure will be considered to be valid only if it first occurs after a BONCE time has elapsed since the last key closure. This prevents double key entries from being recognized and causing errors. Because BONCE and the display one-shot combine to make the period of the system on the order of 50—100 ms, the effect should never be noticeable to the user but completely does away with false inputs due to switch bounces or double key depressions.

Once a valid key closure has been found a jump to KBDS6 occurs. This program segment is used to format the key closure data into a form that is directly usable by the calling program. When this point in the program is reached it is because either the step one-shot is outputting a pulse or one of the mode or hex number keys has been depressed and the KBDS5 program segment has found the key closure to be valid. All of these cases involve only bit positions 0, 1, 2 and 3. The problem now becomes one of introducing the three LSB of the number key into the accumulator in the correct bit positions. The three LSB are, in fact, the contents of the latch driving the decoder circuits that scan the keyboard. This latch is duplicated in the D register which, together with the E register, are acting as a pointer to the keyboard rows.

At the sametime we must somehow reduce the number of bit positions occupied by the number key columns from two to one. These columns after all represent the most significant bit of a four bit hexadecimal number. The three LSB will be supplied from the keyboard row decoder. The information for the fourth, MSB, is contained in the two keyboard columns; obviously there can only be one MSB so the two columns will have to be compressed into one.

Of the four bits in the accumulator of concern, one has to be a logic one or this point in the program would never have been reached. Just as important is the fact that there can only be one logic one in the accumulator at this point. What we are about to suggest may seem an unlikely solution but after examining each of the possible cases we think you'll agree the method of compressing the two column bits into one is easy to say the least. Simply drop the least significant column bit! After all, if the one in the accumulator is due to the step one-shot or one of the mode keys, there will be zeroes in both column bit positions and we can drop the least significant column with impunity. If the logic one is due to the most significant column bit, the least significant column will be zero anyway so we can still drop it. But what if the one is due to the least significant column itself? We can still drop it since the second column bit will be zero! If we use this second column bit for the

most significant bit of the four bit hexadecimal number, it contains all the data needed without using the least significant column bit. If this seems confusing, remember that this state of affairs is only possible because we are not dealing with the full range of binary numbers here since there can never be more than one logic one in the accumulator. Hence we never have to concern ourselves with the possibility of both column bits being one.

Once the decision to drop the least significant column bit has been made, we begin to process the key closure by zeroing out the rest of the bits in the accumulator with an ANI 3EH. After executing this instruction only the two DIP switch inputs, the step one-shot bit, the mode key column bit and the most significant of the two number column bits remain. The least significant column bit along with the other accumulator bits have been zeroed out. Dropping the least significant column bit leaves the least significant bit position of the accumulator vacant. We then execute two RLC instructions to rotate the accumulator two bit positions to the left. This opens up two additional vacant bit positions in the least significant end of the accumulator. Along with the one resulting from dropping the column bit this leaves the accumulator with a total of three vacancies at the right. By moving the contents of the accumulator into the E register, MOV E,A, the accumulator is duplicated in the E register. Here it will be safely out of the way while we manipulate the contents of the D register. D, as you remember, is acting as the pointer that scans the keyboard row. MOV A,D brings it to the accumulator and ANI 0EH zeroes out any bits other than the three that select the keyboard row. An RRC rotates the accumulator one bit to the right so that the row code occupies the three least significant bit positions. The row code is in the accumulator and the column code is in register E along with the image of the mode key column and the step one-shot. ORA E marries the two and leaves the result in the accumulator. If the key closure was one of the hexadecimal keys, the four bit binary value of that key is in the four least significant bit positions; the rest are zero. If the key was one of the mode keys, bit position #4 contains a one and the three LSB positions contain the code to determine which of the mode keys caused the jump. If the jump to KBDS6 was due to the output of the step one-shot, all of the bits except that in position #5 are zero. Finally, if one of the DIP switches caused the jump, that information is in bit positions #6 and #7. All that remains is to POP the various registers to restore their former contents and return to the calling program.

MODE CHANGE CHECK

Many of the monitor programs call upon the keyboard display subroutine to display the contents of the display buffers and to scan the keyboard for key closures. When the subroutine is exited it does so in the form of a return to the calling program. At the time of the return the image of the key closure is in the accumulator. This key closure may be the result of an output pulse from the step one-shot, one of the mode keys or one of the hexadecimal keys. Naturally the system will treat the step one-shot or mode key closures considerably differently from the way it treats a hexadecimal number. In the case of the DCM mode, for instance, the KBDS subroutine was called in the expectation that when a return from the subroutine occurred there would be the image of a hexadecimal number in the accumulator. All of the subsequent program steps were based upon this assumption.

But what would happen if the accumulator in fact did contain, not a hexadecimal number, but the image of a mode key closure or the output pulse of the step one-shot? In both of these cases there should be an immediate exit from the DCM program and a jump made to the new mode as called for by the key closure. This possibility requires us to follow KBDS by a call for the subroutine MDCCK, Mode Change Check. This subroutine is executed as soon as a return from KBDS occurs. It serves to determine if the key image in the accumulator belongs to the step one-shot or one of the mode or hex keys. This subroutine is unusual in the sense that it can be exited in two ways. If the key image in the accumulator turns out to be that of one of the hexadecimal keys the exit is via a normal return to the calling program. In this mode, the MDCCK subroutine does not change the contents of the accumulator. The accumulator contained the image of a hexadecimal number when the subroutine was called; when the return from the subroutine occurs the same number is in the accumulator. If, however, the accumulator contains a bit from the step one-shot or from one of the mode keys the subroutine is exited via a jump to the appropriate program segment.

This dual exit poses some problems. When the MDCCK subroutine is called

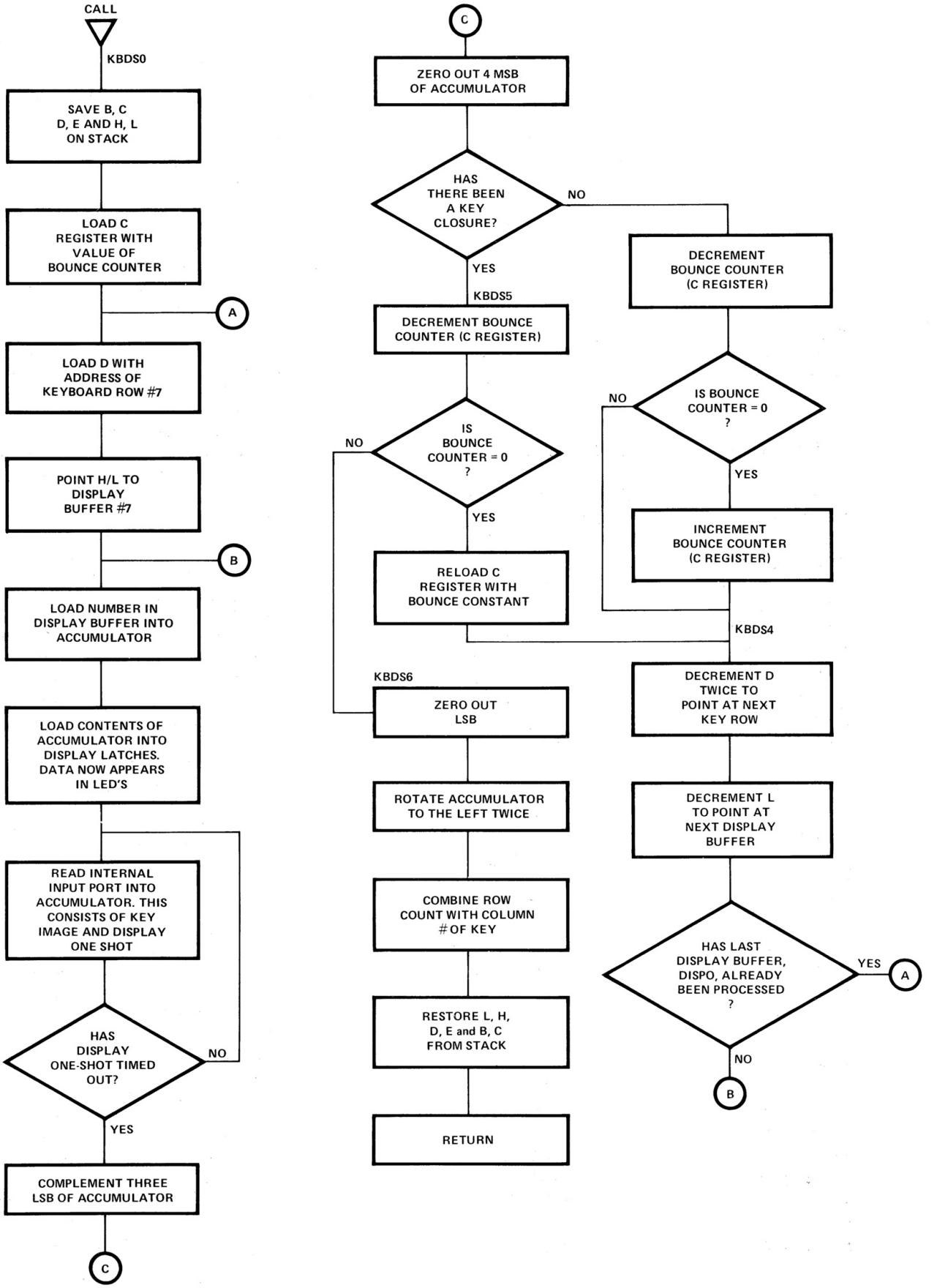

CALL
KBDS0
SAVE B, C D, E AND H, L ON STACK
LOAD C REGISTER WITH VALUE OF BOUNCE COUNTER
A
LOAD D WITH ADDRESS OF KEYBOARD ROW #7
POINT H/L TO DISPLAY BUFFER #7
B
LOAD NUMBER IN DISPLAY BUFFER INTO ACCUMULATOR
LOAD CONTENTS OF ACCUMULATOR INTO DISPLAY LATCHES. DATA NOW APPEARS IN LED'S
READ INTERNAL INPUT PORT INTO ACCUMULATOR. THIS CONSISTS OF KEY IMAGE AND DISPLAY ONE SHOT
HAS DISPLAY ONE-SHOT TIMED OUT?
NO
YES
COMPLEMENT THREE LSB OF ACCUMULATOR
C
C
ZERO OUT 4 MSB OF ACCUMULATOR
HAS THERE BEEN A KEY CLOSURE?
NO
YES
KBDS5
DECREMENT BOUNCE COUNTER (C REGISTER)
IS BOUNCE COUNTER = 0 ?
NO
YES
RELOAD C REGISTER WITH BOUNCE CONSTANT
KBDS6
ZERO OUT LSB
ROTATE ACCUMULATOR TO THE LEFT TWICE
COMBINE ROW COUNT WITH COLUMN # OF KEY
RESTORE L, H, D, E and B, C FROM STACK
RETURN
DECREMENT BOUNCE COUNTER (C REGISTER)
IS BOUNCE COUNTER = 0 ?
NO
YES
INCREMENT BOUNCE COUNTER (C REGISTER)
KBDS4
DECREMENT D TWICE TO POINT AT NEXT KEY ROW
DECREMENT L TO POINT AT NEXT DISPLAY BUFFER
HAS LAST DISPLAY BUFFER, DISPO, ALREADY BEEN PROCESSED ?
YES
A
NO
B

FIG. 6-21 Flow diagram for the KBDS subroutine.

the contents of the program counter are automatically pushed onto the stack memory to provide a return address when the subroutine is completed. At the same time the stack pointer is automatically decremented by two to correspond to the fact that the stack has grown downward by two bytes due to the savings of the program counter. This is all well and good when the subroutine is exited by a return to the calling program. The program counter is popped off the stack to provide a return address and the stack pointer is incremented by two. But when a mode change is found and a jump to the new mode program made, the stack still contains the image of the old program counter and the stack pointer still points to the as yet unused new location in the stack memory. It is easy to see that every time MDCCK finds a new mode key depressed, the stack memory will grow by another two bytes. This violates our rule of a POP for every PUSH to keep the stack size constant. We will consider the solution in a moment.

As a first step of MDCCK the contents of the accumulator are duplicated in the E register with a MOV E,A. This having been done we go on to solve the problem of the ever growing stack memory. The solution is as follows: the MDCCK subroutine must end with a RET instruction to return program control to the calling program. When that instruction is executed the stack pointer will be incremented by two to allow for the fact that the program counter is popped off the stack. If the alternative method of exiting the subroutine is to be a jump, that jump instruction must occur in the program before the RET instruction. Obviously the jump will have to be some sort of conditional jump; otherwise the subroutine would always exit in this mode. If the jump occurs that stack pointer will be too small by two bytes; that is, the program counter will not be popped off the stack in this exit. We can corrrect the problem by incrementing the stack pointer twice before the jump instruction is reached. That way the stack pointer will be pointing to a location two bytes above the program counter. Since the program counter will not be popped off the stack during a jump instruction this corrects the stack size.

But what if the jump test fails and the system goes on to execute the RET instruction? Now the fact that we incremented the stack pointer earlier in the program will cause an error when the system pops two bytes off the stack assuming they are the value of the program counter. To allow for this possibility we must decrement the stack pointer twice after the jump instruction. That way if the conditional jump does not occur the stack pointer is adjusted back to its original value so the RET instruction will execute properly.

The MDCCK subroutine is illustrated in Figs. 6-23 and 6-24. Here we see the MOV E,A to duplicate the image of the accumulator in the E register and the two INX SP instructions to increment the stack pointer to precondition it in case the exit occurs in the jump mode. The next problem is to determine whether

```
80F2 C5        KBDS0: PUSH B
80F3 D5               PUSH D
80F4 E5               PUSH H
80F5 0E32             MVI C, BØNCE

80F7 16CF      KBDS1: MVI D, KERØ7
80F9 210700           LXI H, DISP7
80FC 7E        KBDS2: MØV A,M
80FD 12        KBDS3:  STAX D
80FE 1A               LDAX D
80FF E640             ANI 40H
8101 C2FE80           JNZ KBDS3+1
8104 1A               LDAX D
8105 EE07             XRI 07H
8107 E60F             ANI 0FH
8109 C21A81           JNZ KBDS5
810C 0D               DCR C
810D C21181           JNZ KBDS4
8110 0C               INR C

8111 15        KBDS4: DCR D
8112 15               DCR D
8113 2D               DCR L
8114 FAF780           JM KBDS1
8117 C3FC80           JMP KBDS2
811A 0D        KBDS5: DCR C
811B CA2481           JZ KBDS6
811E 0E32             MVI C, BØNCE
8120 C31181           JMP KBDS4
8124 E63E      KBDS6: ANI 3EH
8126 07               RLC
8127 07               RLC
8128 5F               MØV E,A
8129 7A               MØV A,D
812A E60E             ANI 0EH
812C 0F               RRC
812D B3               ØRA E
812E E1               PØP H
812F D1               PØP D
8130 C1               PØP B
8131 C9               RET
```

FIG. 6-22 Program listing of KBDS routine.

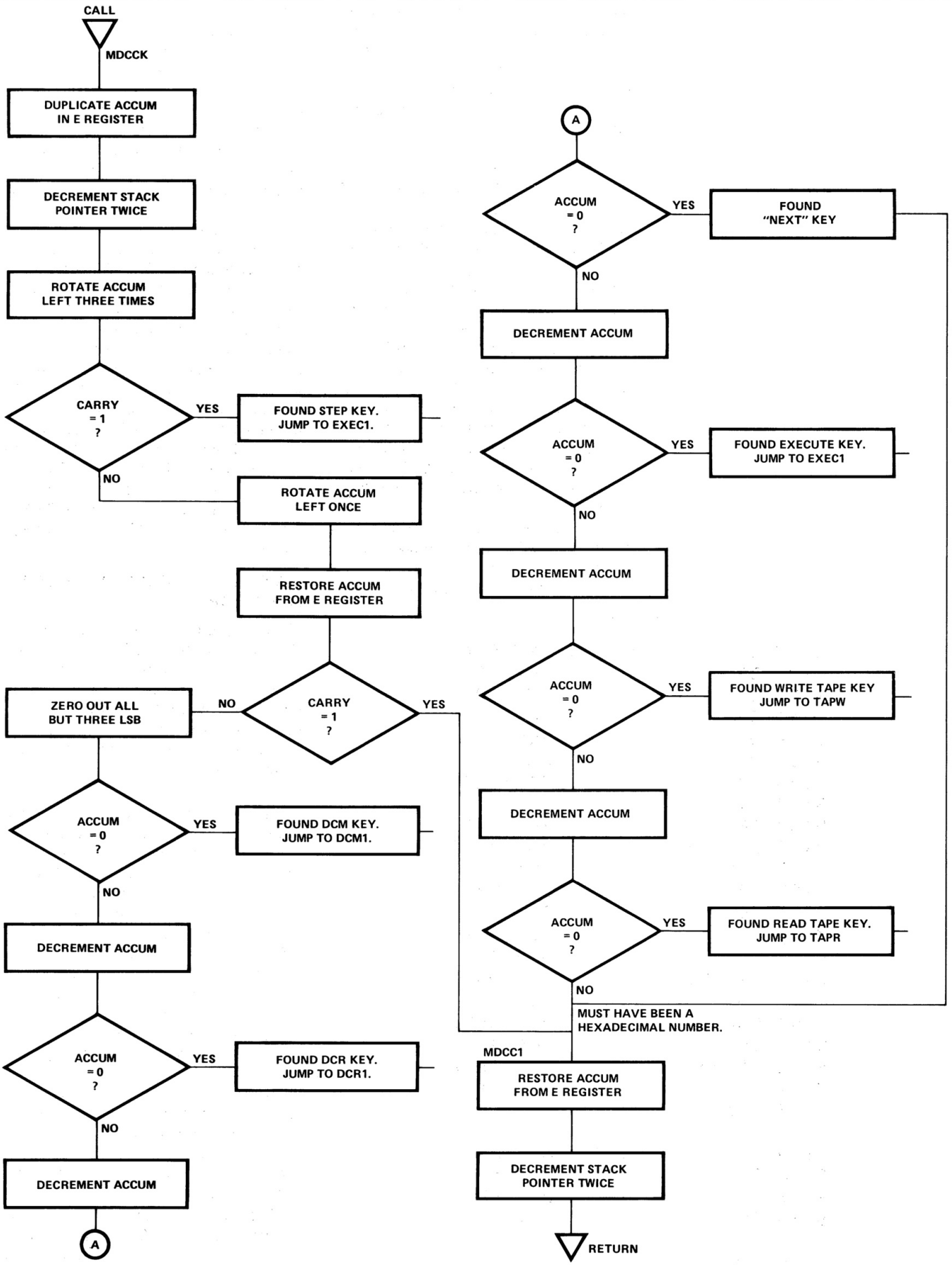


FIG. 6-23 Flow diagram of MDCCK.

the step one-shot has caused the exit from the KBDS subroutine. If the one-shot delivered a pulse it will now be in the bit position #5 of the accumulator. Executing three RLC instructions will rotate this bit, if present, into the carry bit associated with the accumulator. Once here, the bit can be tested with a JC EXEC1 instruction which causes a jump to EXEC1 if the bit is present. If not, the test will fail and the system will pass on to the next instruction, another RLC. This serves to rotate the bit that was originally in bit position #5, the mode key column, into the carry bit. A bit here indicates that one of the mode keys was depressed. The problem now is to find out which one. If this bit is not present the key closure must have been one of the hexadecimal keys and a jump to MDCC1 occurs. MDCC1 is the end of the subroutine; it decrements the stack pointer twice, recreates the original accumulator image by MOV A,E and executes a RET to return to the calling program.

If one of the mode keys was depressed the jump to MDCC1 will not occur. We recreate the accumulator image by MOV A,E and then zero out all but the three LSB positions. All that is then left in the accumulator is the three bit code that determines the keyboard row. The ANI 07H, besides zeroing certain of the accumulator bits, also sets the flags. The next instruction, JZ DCM1, causes a jump to the DCM program segment if the DCM key was depressed (the DCM key corresponds to row #0 of the keyboard). If this test fails, one of the other mode keys must have been depressed. DCR A decrements the accumulator. If the key in row #1, the DCR key, had been depressed the DCR A instruction would now zero the accumulator and set the zero flag. The JZ DCR1 instruction tests the zero flag and causes a jump to the DCR program segment if it is set. Another DCRA decrements the accumulator again. If the key in row #2, NEXT, was set, this instruction will zero the accumulator. JZ MDCC1 causes a jump to MDCC1 which causes an exit from the program with the image of NEXT in the accumulator. In a similar fashion conditional jumps to EXEC1, Execute, TAPR, Read Tape, and to TAPW, Write Tape, are made.

```
814E 5F        MDCCK: MØV E,A
814F 33               INX SP
8150 33               INX SP
8151 07               RLC
8152 07               RLC
8153 07               RLC
8154 DA0182           JC EXEC1
8157 07               RLC
8158 7B               MØV A,E
8159 D27581           JNC MDCC1
815C E607             ANI 07H
815E CA2180           JZ DCM1
8161 3D               DCR A
8162 CA7981           JZ DCR1
8165 3D               DCR A
8166 CA7581            JZ MDCC1
8169 3D               DCR A
816A CA0182           JZ EXEC1
816D 3D               DCR A
816E CA9B82            JZ TAPW
8171 3D                DCR A
8172 CA1783            JZ TAPR

8175 7B        MDCC1: MØV A,E
8176 3B               DCX SP
8177 3B               DCX SP
8178 C9               RET
```

FIG. 6-24 Program listing of MDCCK routine.

DISPLAY AND CHANGE REGISTER MODE, DCR

The DCR mode is entered whenever the DCR key is depressed. This is detected by the KBDS subroutine and entered from the MDCCK subroutine. In this mode all of the working registers of the 8080 microprocessor are made available to the user; he may examine and change them at will.

When the mode is first entered the system displays the letter A, for accumulator, and the contents of the accumulator. If the user wishes to change the contents of the accumulator he keys in the new data through the keyboard. When he is satisfied with his entry as displayed on the LEDs, he presses the NEXT key. The data on the display is loaded into the accumulator and the next register, PSW is displayed. PSW represents the status flags of the CPU. These include zero, sign, carry, auxiliary carry and parity. Again, the user may change this register by keying in new data. Depressing NEXt will call up the contents of the B register which may then be changed in the same fashion. In this manner all of the registers may be examined and modified including the stack pointer and program counter.

The first portion of the DCR program is illustrated in Figs. 6-25 and 6-26. The program is begun by initializing all of the displays with dashes. This can be done simply by loading the dash symbol into

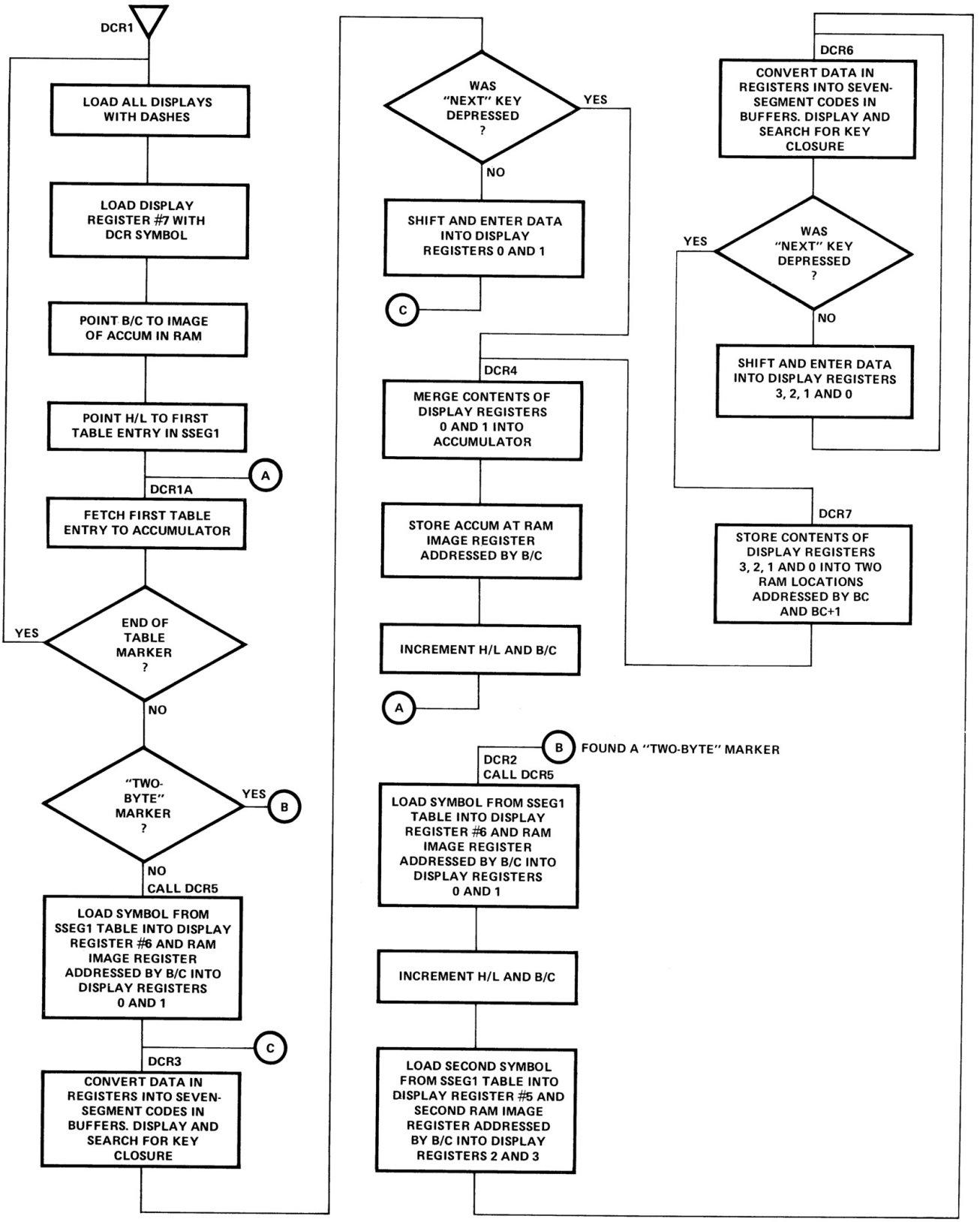

DCR1
LOAD ALL DISPLAYS WITH DASHES
LOAD DISPLAY REGISTER #7 WITH DCR SYMBOL
POINT B/C TO IMAGE OF ACCUM IN RAM
POINT H/L TO FIRST TABLE ENTRY IN SSEG1
A
DCR1A
FETCH FIRST TABLE ENTRY TO ACCUMULATOR
END OF TABLE MARKER ?
YES
NO
"TWO-BYTE" MARKER ?
YES
B
NO
CALL DCR5
LOAD SYMBOL FROM SSEG1 TABLE INTO DISPLAY REGISTER #6 AND RAM IMAGE REGISTER ADDRESSED BY B/C INTO DISPLAY REGISTERS 0 AND 1
C
DCR3
CONVERT DATA IN REGISTERS INTO SEVEN-SEGMENT CODES IN BUFFERS. DISPLAY AND SEARCH FOR KEY CLOSURE
WAS "NEXT" KEY DEPRESSED ?
YES
NO
SHIFT AND ENTER DATA INTO DISPLAY REGISTERS 0 AND 1
C
DCR4
MERGE CONTENTS OF DISPLAY REGISTERS 0 AND 1 INTO ACCUMULATOR
STORE ACCUM AT RAM IMAGE REGISTER ADDRESSED BY B/C
INCREMENT H/L AND B/C
A
B
FOUND A "TWO-BYTE" MARKER
DCR2
CALL DCR5
LOAD SYMBOL FROM SSEG1 TABLE INTO DISPLAY REGISTER #6 AND RAM IMAGE REGISTER ADDRESSED BY B/C INTO DISPLAY REGISTERS 0 AND 1
INCREMENT H/L AND B/C
LOAD SECOND SYMBOL FROM SSEG1 TABLE INTO DISPLAY REGISTER #5 AND SECOND RAM IMAGE REGISTER ADDRESSED BY B/C INTO DISPLAY REGISTERS 2 AND 3
DCR6
CONVERT DATA IN REGISTERS INTO SEVEN-SEGMENT CODES IN BUFFERS. DISPLAY AND SEARCH FOR KEY CLOSURE
WAS "NEXT" KEY DEPRESSED ?
YES
NO
SHIFT AND ENTER DATA INTO DISPLAY REGISTERS 3, 2, 1 AND 0
DCR7
STORE CONTENTS OF DISPLAY REGISTERS 3, 2, 1 AND 0 INTO TWO RAM LOCATIONS ADDRESSED BY BC AND BC+1

FIG. 6-25 Flow diagram of DCR.

the accumulator with a MVI A, DASH followed by a CALL SUB5 which loads each of the displays with the contents of the accumulator. This is followed by loading the DCR symbol into display register #7. The H/L registers are then pointed to display register #7.

Located in PROM is a symbol table made up of the various 7-segment codes corresponding to the letters and symbols used by the system monitor. The first entry in this table is the 7-segment code for the number 0. The second is 1, and so on. We will use this table during the CONV1 subroutine that loads the display buffers with 7-segment codes corresponding to the data in the display registers. We will use the table in this mode during the DCR routine every time CONV1 is called. We will, however, use it in a second way to display the letters that identify the various registers, A, B, C, etc. In this mode, both letters and hexadecimal numbers will appear in the displays at the same time.

The CONV1 subroutine operates by using the hexadecimal number in the display registers to indicate how many table entries from the beginning of the table the correct code will be found. Thus the code for 0H will be in table entry #0, the code for 1H will be in table entry #1, the code for CH will be in table entry #C, etc. After the 0-F entries, the rest of the table is made up of the dashes, mode symbols and odd letters that are used in the rest of the monitor.

Because CONV1 will be called to convert the data in the display registers to 7-segment codes, the progress of loading some of the display registers with symbols must be done in a way that will be compatible with CONV1. Of course, we could use a MVI instruction to load each of the display registers with the correct offset so that CONV1 will fetch the correct symbol from the code table. This method has the disadvantage of requiring a separate MVI for each of the symbols in the code table, which of course removes the possibility of using loops and makes DCR prohibitively long. Instead we will set up a second table that consists of the offsets required for the first table. These are entered in the table in the order they will be required so that DCR can be made up of loops and shortened considerably.

Once the dashes and DCR symbols have been loaded into the display registers, the B/C registers are pointed to the first RAM location holding the images of the CPU registers. This is the image of the accumulator as it was when the user's program was exited. H/L are pointed to the first entry in the table of offsets in PROM. The first entry corresponds to the accumulator image and hence is the offset for the letter A. Since A is one of the hexadecimal numbers, its offset is in fact 0AH. Thus our DCR routine should cause the letter A to be displayed along with the image of the accumulator as it has been stored in RAM.

Before the RAM is accessed for the image of the accumulator we need to do some tests. We want to be sure that the register is not one of the 16-bit registers, stack pointer or program counter. We will want to also test to see if we are at the end of the table. We do this by inserting some markers into the table of offsets, a 0 at the end of the table and a 1 before the first offset of a two byte register. To perform the tests we fetch the first offset from the table with a MOV A,M. This instruction by itself does not set the flags, so an ORA A is inserted. This does not affect the contents of the accumulator at all, but it does set all of the flags. The contents of the accumulator can then be tested to see if they are zero or not. This is done with a JZ DCR1. Thus if the offset fetched from the offset table is a 0 indicating that end of the table has been reached, a jump back to the beginning of DCR occurs, the registers are all reinitialized and the process begins again.

Assuming the end of table test fails, we will go on to test for a 2-byte marker. These consists of 1's preceding the first offset for a 2-byte register. Since H/L are still pointing to the offset just tested we could use them to fetch the offset again for the second test. There is an easier way. DCR M causes the offset to be fetched (using H/L as a pointer), decremented and placed back in the same location. Since this table is contained in PROM, the process of writing the decre-

```
8179 3E10      DCR1:  MVI A, DASH
817B CDCE80           CALL SUB5
817E 210F00           LXI H, DIRG7
8181 3613             MVI M, DECER
8183 011000           LXI B, ACIMG
8186 217382           LXI H, SSEG1
8189 7E        DCR1A: MØV A,M
818A B7               ØRA A
818B CA7981           JZ DCR1
818E 35               DCR M
818F CAB281           JZ DCR2
8192 CDF881           CALL DCR5
```

FIG. 6-26 Program listing for DCR1 and DCR1A.

mented number back into memory fails. The process of decrementing the number itself takes place in the accumulator of the 8080 and in so doing the flags are set. We can use these flags to test for the 2-byte marker, 1. If the offset fetched was a 1, the decrementing process will make it a 0 and JZ DCR2 will cause a jump to the 2-byte program segment. If not, the test fails and the program execution falls through to the next instruction which is a CALL DCR5.

DCR5, see Fig. 6-27, is used to load the offset into display register #6 and the contents of the RAM image register into display registers 0 and 1. The subroutine begins by loading the offset into the accumulator with a MOV A,M. STA DIRG6 then loads this directly into display register #6. LDAX B loads the accumulator with the contents of the RAM location addressed by B/C. During the first pass through the DCR program, this will be the RAM location that serves as the image register for the accumulator. SUB3A is then CALLed to load the contents of the accumulator into display registers 0 and 1.

To review what has happened so far, the displays were loaded with the dashes and DCR symbol. The first offset out of the offset table was loaded into display register #6; when CONV1 is called this offset will become the 7-segment code for the letter A. Also the contents of the RAM loation that serves as the image register for the accumulator loaded into display registers 0 and 1. When the actual display occurs, the user will see the DCR symbol in display #7, the letter A in display #6 and the contents of the user's accumulator in displays 0 and 1. The latter are displayed in hexadecimal.

When the return from DCR5 occurs it will be at the next instruction which happens to be the beginning of DCR3. This is the familiar CALL CONV1, CALL KBDS0 and CALL MDCCK which are used to perform the 7-segment conversions and search for key closures. KBDS0 is an endless loop that is not exited until a key closure is found. At that time it is necessary to determine if the NEXT key was depressed. This is done with CPI NEXT followed by a JZ DCR4. If the key closure was not NEXT, the JZ test fails and SUB1A is called to shift the data into display registers 0 and 1. This is followed by a JMP to DCR3 which sets up another loop. As long as the user confines himself to the hexadecimal keys, he can enter and shift in data into the display registers 0 and 1 as long or often as he wishes. As soon as he is satisfied with the display, he depresses NEXT which causes an exit from the loop to DCR4. See Fig. 6-28.

The DCR4 segment, see Fig. 6-29, causes the contents of display registers 0 and 1 to be loaded back into the RAM location addressed by B/C. This is done with a CALL SUB2A, which loads the two display registers into the accumulator, and a STAX B which stores the accumulator at the location addressed by B/C. Both H/L and B/C are then incremented and a jump back to DCR1A executed to begin the process over again. This time the offset fetched from the table will be for the letter F, for Flags, and the RAM register will contain the image of the accumulator flags. This process continues, displaying first A for accumulator, F for the flags, then the B, C, D, E, L and H registers. After the H register, the stack pointer will be displayed which requires both a two letter symbol, SP, and 4 digits for the 16-bit contents of the register. This has been allowed for by placing the 2-byte marker, 1, before the entry for the first of the two symbols. When this marker is fetched, the test in DCR1A causes a jump to DCR2 which will service the 2-byte registers.

The program for servicing the 2-byte registers is illustrated in Fig. 6-30. It begins at DCR2 with an INX H so that the memory pointer skips over the 2-byte marker to the first of the real offsets. DCR5 is then CALLed to load the symbol into display register #6 and the first of the RAM image registers into display registers 0 and 1. This portion is exactly as it was during the single byte registers. The segment goes on to increment H/L and B/C. The second symbol is fetched with MOV A,M and loaded into display register #5 with a STA DIRG5. The second byte of image register is loaded into the accumulator with LDAX B and then loaded into display registers 2 and 3 by CALLing SUB3. DCR6 causes the contents of the display registers to be converted into 7-segment codes, displayed on the LEDs and key closures searched for. Again, the presence of a NEXT key closure is checked with CPI NEXT. If no NEXT is found the data is shifted into display registers 3, 2, 1 and 0 with SUB1C and a return

```
81F8 7E        DCR5:  MØV A,M
81F9 320E00           STA DIRG6
81FC 0A               LDAX B
81FD CD9A80            CALL SUB3A
```

FIG. 6-27 Program listing for DCR5.

```
8195 CD3281    DCR3:  CALL CØNV1
8198 CDF280           CALL KBDSO
819B CD4E81           CALL  MDCCK
819E FE12             CPI NEXT
81A0 CAA981           JZ DCR4
81A3 CD6880           CALL SUB1A
81A6 C39581           JMP DCR3
```

FIG. 6-28 Program listing for DCR3.

```
81A9 CD8C80    DCR4: CALL SUB2A
81AC 02               STAX B
81AD 23               INX H
81AE 03               INX B
81AF C38981           JMP DCR1A
```

FIG. 6-29 Program listing for DCR4.

```
81B2 23        DCR2:  INX H
81B3 CDF881           CALL DCR5
81B6 23               INX H
81B7 03               INX B
81B8 7E               MØV A,M
81B9 320D00           STA DIRG5
81BC 0A               LDAX B
81BD E5                PUSH H
81BE C5                PUSH B
81BF 210B00            LXI H, DIRG3
81C2 0E02              MVI C, 02H
81C4 CDA580            CALL SUB3
81C7 C1                PØP B
81C8 E1                PØP H
81C9 CD3281    DCR6:  CALL CØNV1
81CC CDF280           CALL KBDSO
81CF CD4E81           CALL MDCCK
81D2 FE12             CPI NEXT
81D4 CAE681           JZ DCR7
81D7 E5               PUSH H
81D8 C5               PUSH B
81D9 210B00           LXI H, DIRG3
81DC 0E04             MVI C, 04H
81DE CD7C80           CALL SUB1C
81E1 C1                PØP B
81E2 E1                PØP H
81E3 C3C981            JMP DCR6
81E6 CD8C80    DCR7: CALL SUB2A
81E9 0B               DCX B
81EA 02               STAX B
81EB E5               PUSH H
81EC 210B00           LXI H, DIRG3
81EF CD9080           CALL SUB2
81F2 E1               PØP H
81F3 03               INX B
81F4 02               STAX B
81F5 C3A981           JMP DCR4
```

FIG. 6-30 Program listing for DCR2, DCR6 and DCR7.

to DCR6 which causes the program to loop as data is entered and shifted. When the user is satisfied with the display he presses NEXT which causes a jump to DCR7 where the contents of the display registers are loaded back into the appropriate image registers. The program is completed with a jump to DCR4.

EXECUTE PROGRAM, EXC

When the user is satisfied with the contents of the memory locations containing his program as determined by the user of the DCM mode and the registers as determined by the use of the DCR mode it is time to try out the program. The user does this by pushing the EXC key. The program that does this is illustrated in Figs. 6-31 and 6-32.

This program mode is one of the few that do not produce a display on the LEDs. To assist the user this is provided for by lighting a discrete LED in the lower left of the board. The address of the port addressing the LEDs is 6000H. Writing a 0F7H to this port will cause the EXECUTE LED to light.

The EXECUTE program is basically one of loading the real registers in the CPU with the images of those registers that are stored in RAM. This must be done very carefully to avoid modifying any of the registers before the execution of the user's program begins. We start the process by loading the stack pointer in the CPU with the stack pointer image in RAM. This is done with an LHLD SPLIM which loads the H and L register pair with the contents of the two RAM locations used to store the image of the user's stack pointer. SPHL then causes the real stack pointer in the 8080 to be loaded with the contents of H and L. Thus the stack pointer has now been correctly loaded.

Next we will set up the program counter. This will actually be the last register loaded but we want to place it on the stack now. This is done with LHLD PCLIM which loads the images of the program counter into the H and L registers. PUSH H then pushes it onto the stack.

With the image of the program counter safely on the stack we will take care of the rest of the image registers by pushing them onto the stack too. This is done with a loop to shorten the amount of coding required. We set up the loop by pointing H/L to the first of the RAM image registers, ACIMG, the location holding the accumulator image. The accumulator itself is loaded with 04H; it will serve as the loop counter.

The loop begins at EXEC2. The first byte of image register is moved into

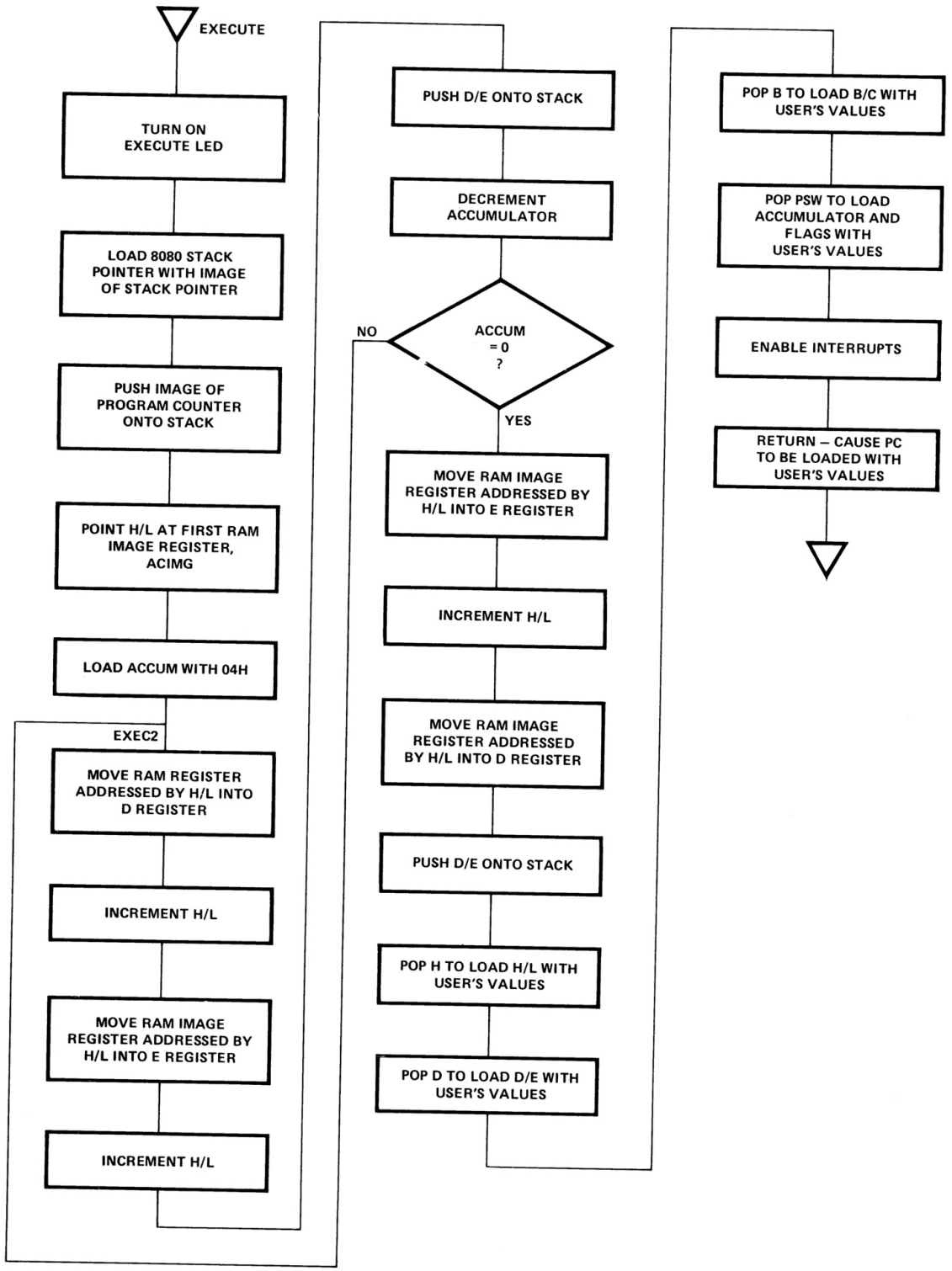

EXECUTE
TURN ON EXECUTE LED
LOAD 8080 STACK POINTER WITH IMAGE OF STACK POINTER
PUSH IMAGE OF PROGRAM COUNTER ONTO STACK
POINT H/L AT FIRST RAM IMAGE REGISTER, ACIMG
LOAD ACCUM WITH 04H
EXEC2
MOVE RAM REGISTER ADDRESSED BY H/L INTO D REGISTER
INCREMENT H/L
MOVE RAM IMAGE REGISTER ADDRESSED BY H/L INTO E REGISTER
INCREMENT H/L
PUSH D/E ONTO STACK
DECREMENT ACCUMULATOR
NO
ACCUM = 0 ?
YES
MOVE RAM IMAGE REGISTER ADDRESSED BY H/L INTO E REGISTER
INCREMENT H/L
MOVE RAM IMAGE REGISTER ADDRESSED BY H/L INTO D REGISTER
PUSH D/E ONTO STACK
POP H TO LOAD H/L WITH USER'S VALUES
POP D TO LOAD D/E WITH USER'S VALUES
POP B TO LOAD B/C WITH USER'S VALUES
POP PSW TO LOAD ACCUMULATOR AND FLAGS WITH USER'S VALUES
ENABLE INTERRUPTS
RETURN – CAUSE PC TO BE LOADED WITH USER'S VALUES

FIG. 6-31 Flow diagram of EXECUTE.

the D register and the second byte moved into the E register. This is followed with a push D which causes this data to be pushed onto the stack. The accumulator is then decremented and tested. If it now contains zero, the number of loops have been implemented and the process stops. Otherwise, the system jumps back to EXEC2 and starts over. This process loads the accumulator, flags, B, C, D and E images onto the stack. The next image registers are the H and L registers; they could be treated as just another loop in the program but for the fact that they are stored in RAM in reverse order. This is taken care of by loading them into D and E in the reverse order the rest of the registers were loaded. A PUSH D then loads these onto the stack. All of the image registers are now contained in the stack. All that remains is to POP H, POP D, POP B and POP PSW to load all of them into the corresponding real registers in the 8080. This leaves only the program counter; firsth though, the interrupts must be enabled with an EI instruction. Otherwise the STEP key which relies on the interrupt capability of the 8080 would not be operable. The final instruction in EXEC is a RET. This causes the last two bytes on the stack memory, which is the image of the user's program counter to be loaded into the real 8080 program counter. As soon as this happens, program execution begins in the user's program at the point determined by the image of his program counter and using the register values as set up by him in the DCR mode.

```
8201 210060    EXEC1: LXI H, 6000H
8204 36F7             MVI M,0F7H
8206 2A1800           LHLD SPLIM
8209 F9               SPHL
820A 2A1A00           LHLD PCLIM
820D E5               PUSH H
820E 210E00           LXI H, ACIMG
8211 3E04             MVI A,04H


8215 56        EXEC2: MØV D,M
8216 23               INX H
8217 5E               MØV E,M
8218 23               INX H
8219 D5               PUSH D
821A 3D               DCR A
821D C21582           JNZ EXEC2
821E 5E               MØV E,M
821F 23               INX H
8220 56               MØV D,M
8221 D5               PUSH D
8222 E1               PØP H
8223 D1               PØP D
8224 C1               PØP B
8225 F1               PØP PSW
8226 FB               EI
8227 C9               RET
```

FIG. 6-32 Program listing for EXECUTE.

SINGLE STEP MODE

When the user sits down at the microcomputer system and attempts to execute his program, the chances of the program running correctly the first time are very slim. The problem quickly becomes one of determining what portion of the program is hanging up the system. In a production microcomputer system that has a lot of I/O ports and peripheral equipment this may be obvious. Usually, the programmer does not try a program out for the first time on this type of system. Instead he will use a development system like the ia7301 that provides him with an easy way to enter and debug the program. At the time of the first program pass there may be few if any of the peripherals tied to the system, so that when the program becomes hung up, it is not always obvious exactly where the problem lies. Many times the only way the user is even aware of the fact that there is a problem is the lack of or incorrect data in the display. The immediate task is then to find the problem area. Only then can he attempt to correct the problem.

The most convenient way to do this is through the use of the single-step mode. When the STEP key is pressed it causes an interrupt which stops the execution of the user's program. The displays will show the user the contents of the program counter at that point as well as the contents of the memory in that location. Thus, the STEP keys stops the user's program wherever it is at that point and shows the user the next instruction to be executed. Each time he presses the STEP key, the system returns to the user's program, executes the instruction just displayed, and displays the next instruction to be executed. At any time, the user may exit the single-step mode to go to the DCR or DCM modes. In this way, he can stop and examine or modify any of the registers or memory locations. When he is satisfied, depressing STEP again will cause the system to execute one instruction and display the next on the LEDs. Obviously the STEP key can make life much simpler for the programmer faced with the task of

debugging a complex new program.

When the STEP key is depressed, an interrupt to the 8080 occurs. Since only one interrupt is allowed with this system, a RST 7 instruction is forced onto the bus. The various RST instructions all jam a different address onto the bus. Since the data bus has pull-up resistors, any line that is not pulled low will load a logic one into the 8080. RST 7 requires all logic one's, so that if this is the RST chosen for the interrupt there is no hardware required to jam the bus.

RST 7 causes the system to jump to location 38H and begin executing instructions at that point. As part of the initialization process at the beginning of the monitor, we load locations 38H, 39H and 3AH with the 3-byte jump instruction, JMP STEP1. This means that whenever the STEP key is pressed, the system jumps to 38H, executes three instructions there (the jump) and then jumps to the first instruction in the STEP program subroutine, STEP1.

When the interrupt occurs, the 8080 pushes the contents of the program counter onto the stack memory and transfers program control to the instruction at 38H. The address saved in the stack is that of the next instruction to be executed. All of the other registers are in the same state they were in when the program was interrupted, that is, they still contain the user's data and have not yet been modified by the monitor. The stack pointer does reflect the fact that the user's program counter has been pushed onto the stack and therefore contains a number two less than it did while running the user's program. The first job of the STEP subroutine is to load the user's registers into the correct locations in RAM memory that are being used as the image registers. See Figs. 6-33 and 6-34.

To do this efficiently we will need to use the H and L registers to point to those RAM locations. But before we can do that we must store the present contents of the H/L pair in their own image registers. This is easily done with a SHLD LIMG instruction which stores both H and L at once. This frees up the H/L register pair.

The next registers to attack are the accumulator and flags. The accumulator is no problem, but the flags are trickier. In fact there is only one instruction that can access all of the flags at once, PUSH PSW. This loads both the accumulator and all of the flags into the stack memory where they are easily accessible to us. If we follow that instructionwith a POP H, the two bytes of data at the bottom of the stack will be loaded into the H and L registers. Hence the combination of a PUSH and a POP succede in loading the accumulator and flags into the H/L pair. SHLD ACIMG then loads these into the image registers in RAM.

The H and L registers are now free to act as a pointer to the image registers for the B, C, D and E registers. LXI H, BIMGE stes up the H/L pointer and MOV M,B loads the B register into its RAM image register. All that remains is to increment L and MOV M,C to load C in the same fashion. The process is repeated for D and E.

Loading the program counter into the image registers is relatively easy. As has already been pointed out, the interrupt caused the program counter to be pushed onto the stack. The first POP that is executed will load this data into the register pair specified in the instruction. In our case this is POP H. The instruction SHLD PCLIM then loads this into the RAM images reserved for the program counter.

Gaining access to the stack pointer is not as easy as the other registers since they are very few instructions that are valid when used with the stack pointer. One of the few that is, DAD SP, operates by adding the contents of the stack pointer to the contents of the H and L register and leaving the 16-bit result in H/L. The POP H that we used to fetch the user's program counter also had the effect of incrementing the stack pointer by two so that it now contains the same value as it did when the user's program was being executed. Thus the effect of LXI H,0 (to clear the H and L registers) and DAD SP is to load the stack pointer current during the user's program into the H and L registers. From there SHLD SPLIM loads the data into the image registers in RAM.

This completes the first task of the STEP subroutine. Before going any further we want to be sure the operator has taken his finger off the STEP key. This is done in STEP1A which reads the output of the step one-shot into the accumulator and loops back to STEP1A until the button is released.

Since STEP is normally entered from EXECUTE we will need to turn off the EXECUTE LED. This is done at the beginning of STEP2. Again the 6000H port is addressed and a write, 0FFH, done to that port. This blanks the LED.

The next problem is to display the contents of the program counter and the instruction that is stored at that location on the displays. MVI A,DASH and CALL SUB5 loads all of the display registers

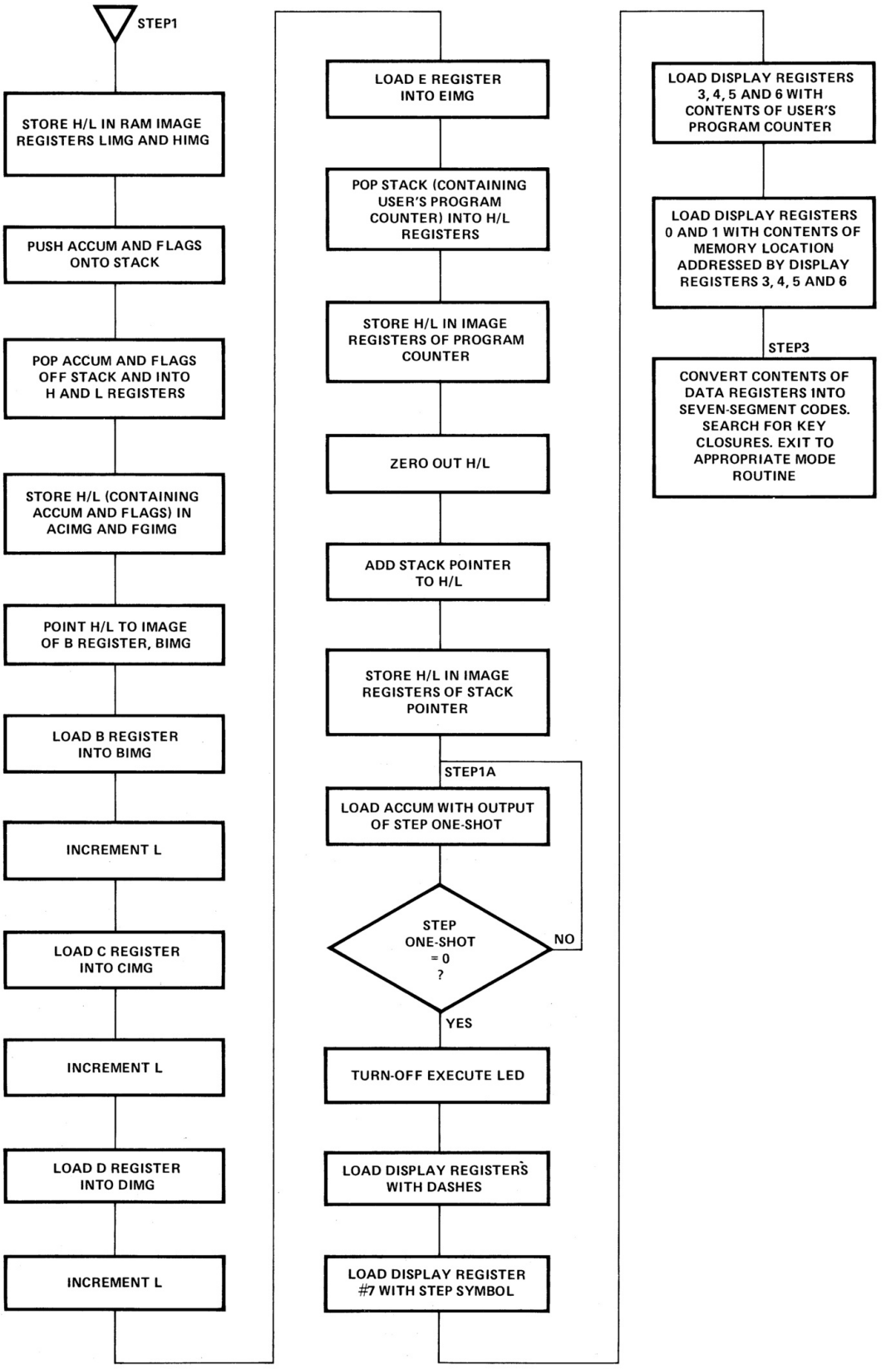

STEP1
STORE H/L IN RAM IMAGE REGISTERS LIMG AND HIMG
PUSH ACCUM AND FLAGS ONTO STACK
POP ACCUM AND FLAGS OFF STACK AND INTO H AND L REGISTERS
STORE H/L (CONTAINING ACCUM AND FLAGS) IN ACIMG AND FGIMG
POINT H/L TO IMAGE OF B REGISTER, BIMG
LOAD B REGISTER INTO BIMG
INCREMENT L
LOAD C REGISTER INTO CIMG
INCREMENT L
LOAD D REGISTER INTO DIMG
INCREMENT L
LOAD E REGISTER INTO EIMG
POP STACK (CONTAINING USER'S PROGRAM COUNTER) INTO H/L REGISTERS
STORE H/L IN IMAGE REGISTERS OF PROGRAM COUNTER
ZERO OUT H/L
ADD STACK POINTER TO H/L
STORE H/L IN IMAGE REGISTERS OF STACK POINTER
STEP1A
LOAD ACCUM WITH OUTPUT OF STEP ONE-SHOT
STEP ONE-SHOT = 0 ?
NO
YES
TURN-OFF EXECUTE LED
LOAD DISPLAY REGISTERS WITH DASHES
LOAD DISPLAY REGISTER #7 WITH STEP SYMBOL
LOAD DISPLAY REGISTERS 3, 4, 5 AND 6 WITH CONTENTS OF USER'S PROGRAM COUNTER
LOAD DISPLAY REGISTERS 0 AND 1 WITH CONTENTS OF MEMORY LOCATION ADDRESSED BY DISPLAY REGISTERS 3, 4, 5 AND 6
STEP3
CONVERT CONTENTS OF DATA REGISTERS INTO SEVEN-SEGMENT CODES. SEARCH FOR KEY CLOSURES. EXIT TO APPROPRIATE MODE ROUTINE

FIG. 6-33. Flow diagram of STEP.

with dashes. Point the H/L pair to display register #7 and load that register with the STEp mode sumbol. We are now ready to fetch the contents of the program counter. This will not be the current program counter since that is presently pointing to some location in the STEP portion of the system monitor. Rather, we wish to display the program counter that was current at the time of the interrupt. This value was saved in the PC image registers in RAM by the initial portion of STEP. An LHLD PCLIM will fetch these 16 bits of data and load them into H and L. At that point we could write a program that would transfer this data into four of the LED displays. Fortunately we have a similar program, SUB6, which loads the contents of the D and E registers into display registers 6, 5, 4 and 3. To use it now all that is necessary is to swap the H/L pair with the D/E pair with an XCHG instruction. CALL SUB6 then causes the address to be displayed.

Fetching the contents of the address being displayed is easy. Since the D/E registers already contain the address, the instruction LDAX D uses D and E as a pointer. The data fetched from the memory is stored in the accumulator. Calling SUB3A causes that data to be displayed in display registers 0 and 1. All that remains is to call CONV1, KBDS0 and MDCCK to display the data in the LEDs. The program loops within KBDS until a key closure is found. MDCCK determines whether or not the key closure is one of the mode keys. If it is, a jump to the appropriate subroutine occurs. If the key closure is from one of the hexadecimal keys the data is ignored. The STEP program concludes with a jump back to CONV1 which causes the program to loop around the display. The loop is exited when MDCCK determines that a mode key has been depressed. Hexadecimal keys have no effect upon the STEP program.

```
8228 221600    STEP1: SHLD LIMG
822B F5               PUSH PSW
822C E1               PØP H
822D 221000           SHLD ACIMG
8230 211200           LXI H, BIMG
8233 70               MØV M,B
8234 2C               INR L
8235 71               MØV M,C
8236 2C               INR L
8237 72               MØV M,D
8238 2C               INR L
8239 73               MØV M,E
823A E1               PØP H
823B 221A00           SHLD PCLIM
823E 210000           LXI H,0
8241 39               DAD SP
8242 221800           SHLD SPLIM
8245 16CF      STEP1A: MVI D,KERØ7
8247 1A                LDAX D
8248 E608              ANI 08H
824A C24582            JNZ STEP1A
824D 210060    STEP2: LXI H,6000H
8250 36FF              MVI M,0FFH
8252 3E10              MVI A,DASH
8254 CDCE80           CALL SUB5
8257 210F00           LXI  H, DIRG7
825A 3614             MVI M, STEP
825C 2A1A00           LHLD PCLIM
825F EB               XCHG
8260 CDDE80           CALL SUB6
8263 1A               LDAX D
8264 CD9A80           CALL SUB3A
8267 CD3281    STEP3: CALL CØNV1
826A CDF280           CALL KBDS0
826D CD4E81           CALL MDCCK
8270 C36782            JMP STEP3
```

FIG. 6-34 Program listing for STEP.

CONVERT

The only remaining job of the system monitor is the conversion subroutine for converting the data in the display registers into 7-segment codes in the display buffers. The conversion subroutine, CONV1, is normally followed by KBDS which then displays the contents of the data buffers in the LEDs and searches for new key closures.

The subroutine begins by saving all of the registers on the stack and pointing B/C to display buffer #7. H/L are then pointed to the first entry in the table of 7-segment codes contained in PROM. This is the first instruction of the CONV2 loop.

LDAX D loads the accumulator with the contents of the display register being addressed by D/E. On the first pass through CONV2 this will be display register #7. To this number, the contents of register L are added. Since H/L were pointed to the starting address of the conversion table, L contains the low order byte of the address of the first entry. The location of the table in PROM has been chosen so that all of the entries can be accessed without having to change the contents of register H. Since the entries are in ascending order, beginning with 0H and ending with FH, adding the data in the accumulator to the contents of L will serve to load the accumulator with the low order byte of the address of the entry that corresponds to the original data. This is then reloaded into

L by MOV L, A and the table entry loaded into the accumulator with MOV A, M. Thus the accumulator now contains the 7-segment code that corresponds to the original data in the display register. STAX B transfers this code to the appropriate display buffer. Registers E and C are then decremented and the latter tested for positive contents. Assuming it does not contain a negative number, a jump back to CONV2 is made. A negative number in register C means that the last of the display buffers has been loaded and the decrementing process took the contents of C which were 00H and produced FFH. When this occurs the conversion routine is completed. All that remains is to restore the registers with a series of POPs and return to the calling program.

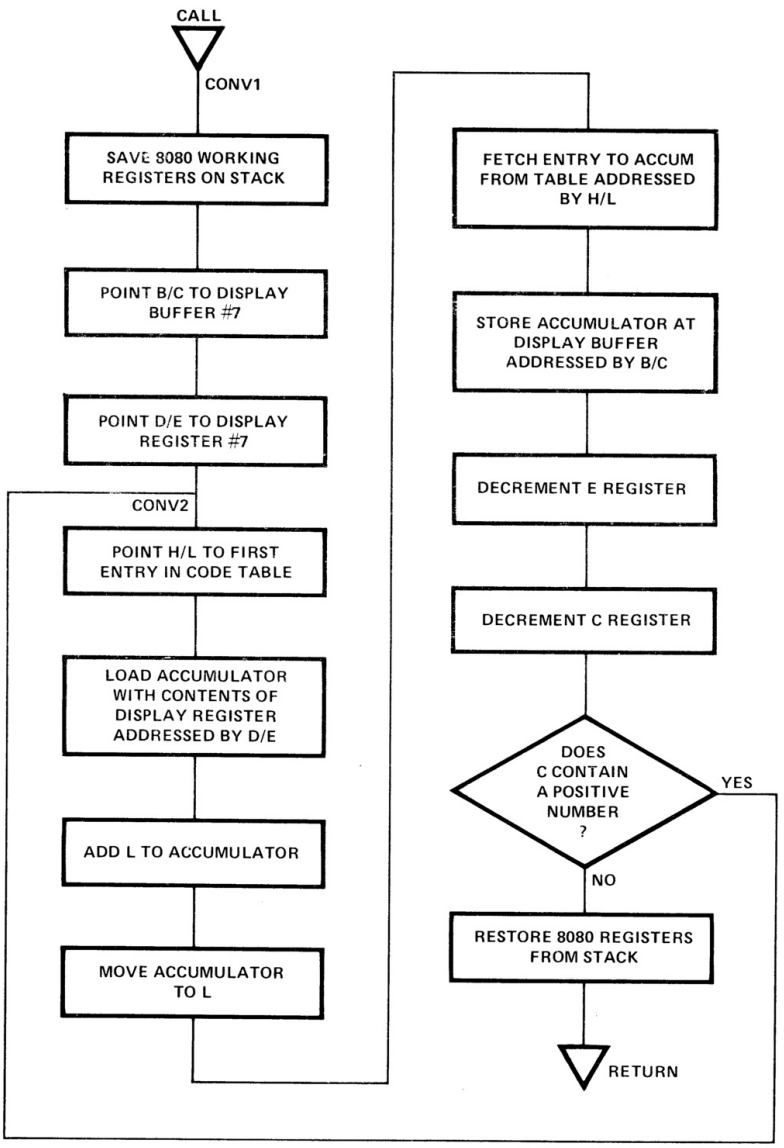

CALL
CONV1
SAVE 8080 WORKING REGISTERS ON STACK
POINT B/C TO DISPLAY BUFFER #7
POINT D/E TO DISPLAY REGISTER #7
CONV2
POINT H/L TO FIRST ENTRY IN CODE TABLE
LOAD ACCUMULATOR WITH CONTENTS OF DISPLAY REGISTER ADDRESSED BY D/E
ADD L TO ACCUMULATOR
MOVE ACCUMULATOR TO L
FETCH ENTRY TO ACCUM FROM TABLE ADDRESSED BY H/L
STORE ACCUMULATOR AT DISPLAY BUFFER ADDRESSED BY B/C
DECREMENT E REGISTER
DECREMENT C REGISTER
DOES C CONTAIN A POSITIVE NUMBER ?
YES
NO
RESTORE 8080 REGISTERS FROM STACK
RETURN

FIG. 6-35 Flow diagram of CONV.

```
8132 C5        CØNV1: PUSH B
8133 D5               PUSH D
8134 E5               PUSH H
8135 F5               PUSH PSW
8136 010700           LXI B, DISP7
8139 110F00           LXI D, DIRG7

813C 218282    CØNV2: LXI H, SSEG4
813F 1A               LDAX D
8140 85               ADD L
8141 6F               MØV L,A
8142 7E               MØV A,M
8143 02               STAX B
8144 1D               DCR E
8145 0D               DCR C
8146 F23C81           JP CØNV2
8149 F1               PØP PSW
814A E1               PØP H
814B D1               PØP D
814C C1               PØP B
814D C9               RET
```

FIG. 6-36 Program listing for CONV.

Chapter Seven
The IASIS ia7600 and ia 7700 Systems

By now it should be painfully obvious to even the most casual reader that designing and debugging a microcomputer system is no easy task. For applications that will require more than a few hundred microcomputer systems, there is no doubt that the rewards are worth the trouble. The microcomputer can bring lower costs, higher performance and greater reliability to many applications. However for those applications requiring less than 100 systems, the cost of developing one of these systems eliminates the potential savings so that the promise of the microcomputer has generally not been realized in low volume applications. This has brought about the introduction of preengineered systems, assembled, tested and ready for use. Even if the decision is made to go the custom route, having a well-documented standard system in operation will cut large amounts of time off the development of the custom design.

Until recently the preengineered systems on the market have been bulky multicard affairs that tended to be very expensive — $2500 and up. At these prices some of the stripped down minicomputers on the market offered more performance at similar prices. Small wonder that until recently preengineered systems have seen little use. Now, however, complete single board systems are becoming available that offer the OEM user a complete operating microcomputer system for less than $500. Unfortunately these have all been designed with little regard to interfacing to a development system. As a result the purchaser of these single board systems has been left largely on his own. With no way to input or assemble programs on the single board computer, the purchase was then left with no choice but to add a full-blown development system. Even in the minimum configuration the development system requires a teletype and a PROM programmer which runs the total system price up to $7500. If the user opts for some of the high speed peripherals the price quickly exceeds $10,000. This is a very stiff price for the new user who isn't really sure that microcomputers represent the best design solution to his application problems.

A second drawback to the present single board computers is in their use of MOS programmable I/O devices. These greatly simplify the manufacturer's problem of which combination of input and output ports he places on the board since the user can program the MOS devices to be either input or output, as he wishes. Unfortunately the characteristics of the MOS process leads to devices with almost no drive capability, very little power noise immunity, and inputs that are susceptible to damage from noise spikes, three very serious drawbacks to a system that has to operate in the industrial world. At least one of these systems uses TTL buffers and resistor pullups to solve the problem of drive capability and to offer some slight protection against noise. This has the disadvantage of requiring a great deal of board space and simply adding resistor pullups to the MOS inputs only increases the zero state noise immunity. It also slightly reduces the danger of damage from spikes although to a degree this danger is present no matter how small the resistors are made.

The most popular of these systems includes a teletype interface and a serial I/O device, a USART, even though the vast majority of these systems will not be used in any kind of a serial I/O mode. This only increases the cost of the system and benefits very few. Most systems that will be used in the serial communications mode will require more memory and I/O than this small system has, so that expander boards will be needed. It is on these boards that the USART and other serial I/O circuits should have been placed so they can be chosen as an option rather than forced on the user who considers them an unnecessary and unneeded expense.

THE IASIS ia7600 OEM SYSTEM

It is with these considerations in mind that the decision was made to bring out a set of Iasis OEM microcomputer boards. The goal was to offer a complete microcomputer on a single board for less than $500. The system, the ia7601 TCS (Total Computer System), was to provide an 8080A, crystal clock, a full set of buffers for system expansion, 1024 bytes of RAM, sockets for 8096 bytes of EROM, and six I/O ports that can be chosen by the user to be interrupting or noninterrupting. The interrupt configuration is a combination of hardware priority interrupts and interrupt polling to allow both high-speed interrupt processing as well as the ability to handle

256 interrupting ports. The I/O consists of two input ports, two output ports and two bidirectional I/O ports that can be used as either inputs or outputs in any combination. Furthermore the choice of input or output is controlled by the system under program control and can be changed during the course of the program itself. All I/O is constructed entirely of TTL circuits, thus avoiding the problems of drive capability, susceptibility to damaging spikes, and noise immunity. The ia7601 is shown in Fig. 7-1.

The memories used on the ia7601 and related boards are high speed static RAMs that eliminate the need for WAIT states. The EROMs are used with 28 pin sockets. This allows the user to specify either 2708 8K EROMs with 24 pin packages or the new electrically erasable 8K EROMs with 28 pin packages. Both devices will use the same socket but the 28 pin part allows the device to be programmed in place without removing it from the socket. It may also be erased, still in the socket, by applying the correct voltages to the system edge connector. This eliminates the need for a separate PROM programmer and ultraviolet light and saves the user a whopping $1500 in development system hardware since the ia7601 now becomes its own PROM programmer.

The ia7601 measures approximately 9" X 12". Two edge connectors are provided, both double position 78 line .125" type. One carries bus signals to the development system and expander daughter boards if used, and the other is the I/O connector for accessing the I/O ports. With the exception of the CPU board, the bus is universal in the sense that all board positions are identical. Thus RAM, EROM, and I/O may be placed anywhere along the bus with no regard for order or position allowing for easy expansion. Most of the daughter boards are half-size, approximately 9" X 6", so that two of them can be placed side-by-side on top of the ia7601. Connector hardware is available from Iasis that allows stacking of the boards to two, three or four levels. With four levels of boards a typical system would contain the CPU, 16K of EROM, 5K of RAM, and 38 I/O ports, 18 of which are bidirectional and programmable. The total package would measure only 9" X 12" X 2". See Fig 7-2.

A variety of signals are brought to the edge connectors. These include EROM and RAM disable lines that greatly simplify the problem of interfacing to a development system. Use of these lines eliminates the need for ROM simulators and In-Circuit Emulators, ICE, that are both

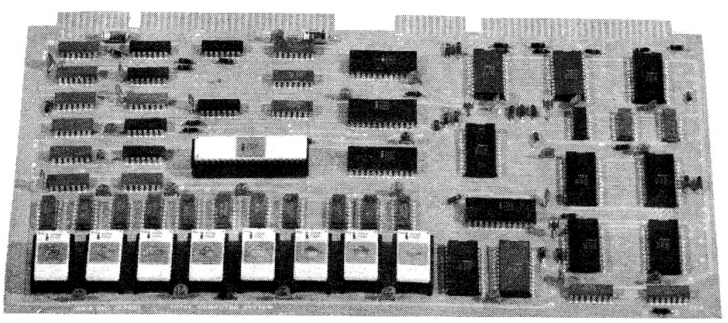


FIG. 7-1. The ia7601 TCS (Total Computer System) is a complete system in itself. It contains an 8080A, crystal clock, 1024 bytes of RAM, sockets for 8192 bytes of EROM and six I/O ports, two of which are bidirectional.

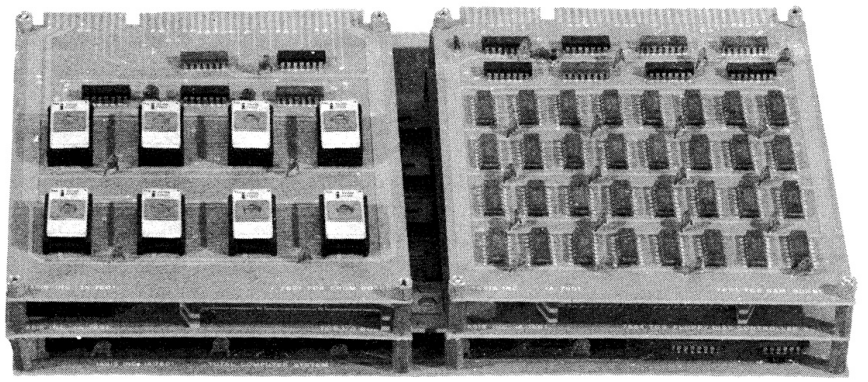


FIG. 7-2. When the 7601 TCS is combined with smaller expander boards, the result is a flexible system that grows with your needs, yet is compact and easy to use. This one has 9216 bytes of RAM, 8192 bytes of EROM and 18 I/O ports.

expensive and now unnecessary. Again, by proper care in the design of the OEM board the user is saved over $5000 in development system costs!

Additional options on the edge connector allow the user to cause an automatic jump to the EROM memory section when a reset is generated. A second option causes the automatic jump to EROM to occur when power is applied also. This eliminates the need for expensive "shadow" PROMs and bootstrap loaders in the development system. It also simplifies the problem of the OEM user who must enter the EROM when the system is reset or energized automatically Another problem common to the industrial user is that of temporary power interruption. The memories chosen for the ia7601 can be either NMOS or pin compatible CMOS devices. These are connected to a separate 5V line on the PC board and brought out to a separate line on the edge connector. If standard NMOS is chosen for the RAMs, the line on the edge connector is tied to the same 5V supply that powers the rest of the system. If, however, CMOS RAMs are selected, the separate 5V line will be tied to a battery supply. In this way, power-downs of a week or more will not cause data loss in the system. This eliminates one of the most serious problems of the OEM user who must contend with the very real problem of losing his data every time there is a power loss, either due to the electric company or to inadvertant handling on the part of plant personnel.

One of the most appealing features of the ia7600 system is the easy way it can fill the dual role of OEM and development system. Adding two RAM expander boards and an EROM expander board with preprogrammed STAGE 1 Monitor program gives the user a complete development system right down to the PROM programmer! In this mode the I/O ports of the ia7601 accept data and instructions and even drive the RAM and EROM disable lines so that the extra RAM on the expander boards fills the role of the ROM simulator. Thus programs are written into RAM occupying the same memory addresses as the EROM on the ia7601 board which has been disabled by the EROM disable line. Then the instructions are programmed into the EROMs on the ia7601 so that the program is in the form of nonvolatile storage. Remove the expander boards and the user now has a completely programmed and operational OEM single board system.

Even at this early stage the user will probably opt for the complete STAGE 1 Kit. This is a kit only in the sense that the user is provided with boards and connectors that he must plug together himself — all of the boards are assembled and tested. The STAGE 1 Kit contains an ia7601 TCS board, two ia7621 4K RAM boards, an ia7631 EROM board containing the STAGE 1 system monitor in EROMs, an ia7661 development board with a complete 66 pad keyboard, cassette tape interface for any consumer tape recorder having earphone and remote microphone jacks, an ia7662 development board containing switches and alphanumeric LEDs for displaying address, data bus, and contents of each of the working registers of the CPU. In addition to the 6 boards connector hardware (prewired), and a full set of cables are provided as well as an inexpensive chassis to hold the boards.

The STAGE 1 Kit provides the user with the ability to enter programs through the ASR-33 style keyboard, and execute and debug programs through the switches, keys and LEDs. An assembler program can be loaded into the system through the tape interface (the assembler is included in the STAGE 1 Kit prerecorded on a cassette tape), allowing the user to write his programs in assembly language.

Future expansion is up to the user, and it comes in a wide variety of flavors. The ia7701 is a complete cabinet enclosure that includes a working 9" video monitor with its own 1024 byte memory (this board, the ia7663, is available separately as well). This provides the user with a complete set of power supplies as well as an attractive enclosure for mounting his boards and keyboard. The video monitor and the ia 7662 board are now mounted in a vertical position behind a sheet of smoked plexiglass. The ia7661 with its keyboard is in a horizontal position for easy typing position. Connectors for a complete system are included so that the user's existing boards can be loaded into the cabinet which includes dual fans for efficient cooling. See Fig. 7-3.

Adding a set of preprogrammed EROMs to the ia7631 board containing the system monitor provides the user with a complete assembler package contained completely in hardware. This eliminates the need for multiple passes of the assembler and source tapes and can save a considerable amount of time. An alternative to this is to add the ia7664 board which is a complete interface to a floppy disk system capable of handling up to four disk drives. A dual disk system, the ia7702, is available in a separate cabinet completely compatible with the ia7600 system. It receives its data and control signals from the ia7664 board which also contains the disk operating system in the form of preprogrammed EROMs. The disks are available in prerecorded form with a complete assembler on them. A second version carries a full BASIC package for programming in the high-level language BASIC. The BASIC package is also available as a set of EROMs designed to occupy the sockets on the ia7601 board. A second tape alternative, the ia7703, consists of a pair of high quality EDP type cassette recorders capable of storing over a half million bytes of data accessible at a rate of 800 baud. These recorders are driven by the ia7665 board which contains all of the interface circuitry and the tape operating system in the form of preprogrammed EROMs. The use of the ia7703 and ia7665 allow the user to store and retrieve programs and data randomly by file management similar to the disk operating system but at a lower cost. For producing hard copy the ia7704 line printer runs at 120 characters per second and is hard to beat. It ties directly to one of the ports on the ia7601; the program for interfacing to the printer (as well as to a standard teletype) are already in the STAGE I monitor. Other expander boards include reed relays, triac switching and opto-isolators, ac-detectors, a double-sized board containing 16K of RAM, as well as a full assortment of standard I/O boards.

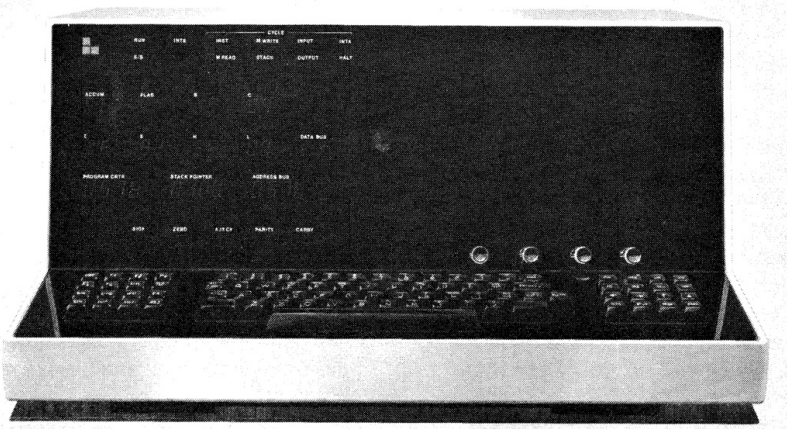


FIG. 7-3. The ia7701 turns the ia7600 OEM boards into a total development system, complete with video monitor, keyboard, LED readouts and interfaces to floppy disks and cassette recorders.

Once a user has made the decision to go with the ia7600 series of OEM and development products the size and versatility of his system is limited only by his imagination. At no time does he ever find himself discarding equipment as he upgrades the system, yet the expansion can take place in small stages that are easy on the budget.

The versatility built into the boards allows them to be used in three general ways; first as an OEM system in either single board form or in a more expanded version with extra memory and I/O, in a development system with keyboard, CRT and assembler programs, or finally, in a pseudo minicomputer with a BASIC package in either EROM or floppy disk. In this mode the system is ideal for either experienced industrial users or hobbyists exploring the world of the computer for the first time.

THE ia7601 TCS BOARD

The lasis ia7601 TCS board is the heart of the entire 7600 OEM system. By itself the board is a complete operating micro-

computer system, but it can be expanded by adding other boards to increase its memory and I/O capabilities. This system has been designed with the industrial user in mind; it offers a high degree of power noise immunity, a versatile I/O structure, easy to use reset features, provisions for power-down data retention, and capacity for over 8000 bytes of EROM on a single board. The system can be broken down into several sections or blocks.

CPU AND BUS BUFFERS

The heart of the ia7601 TCS board is the 8080A microprocessor, capable of executing 78 instructions at a 2 MHz clock rate. This device interfaces to the system through a set of three buses, the address bus, the data bus and the control bus. Addressing 65,536 bytes of memory requires 16 address lines; these are buffered by inverting three-state buffers and brought to the system edge connector. The address signals are represented by $\overline{A_0}$ through $\overline{A_{15}}$. Similarly the data bus consisting of eight signals, $\overline{DB_0}$ through $\overline{DB_7}$ brought to the system edge connector. These too are buffered by inverting three-state bus drivers. Unlike the address buffers the data bus buffers are bidirectional so that data can move in both directions into and out of the 8080. During certain types of operation like DMA activities these buses may be floated so that a peripheral can seize control of memory and I/O functions

CLOCK CIRCUITS

The ia7601 uses a crystal controlled clock. Several signals are brought to the edge connector for use by the development system and expander boards. These include OSC, an 18.432 MHz TTL signal that is a buffered output from the crystal oscillator itself. $\overline{0_2}$ is also brought to the edge connector, although this is not the 12V clock that drives the 8080A but a TTL equivalent with very similar timing. The clock frequency is 2.048 MHz and the pulse width of the 0_2 TTL clock is 235 ns; it lags the falling edge of the 0_1 clock by no more than 15 ns. The $\overline{0_2}$ clock is used by many of the other boards to synchronize the system.

At the beginning of each machine cycle the 8080 sends out a status word on the data bus to be used for the generation of control signals to correctly enable bus buffers, etc. In order to notify the control circuits that the status word is on the data bus, a special signal, $\overline{STSTRB}$ is sent to the control circuits so they can latch and decode the status word. This TTL signal is also sent to the edge connector for use in the development system.

RESET CIRCUITS

When the system is reset, either through use of a reset switch or when the power is first applied, it is necessary to generate a reset pulse to set up the 8080 and to clear the various latches and flip-flops throughout the system. An associated problem is to make the system automatically access the first instruction out of EROM rather than RAM. This could be done by populating the low order memory addresses with EROM but this prevents the user from selecting his own jump addresses for the restart instructions that appear at eight locations in the low order memory. It is much more desirable to implement in hardware an automatic jump to EROM whenever the system is reset. An initialization routine in EROM then sets up the stack pointer and loads the restart addresses with the correct jump addresses. Later, the same program may change the jump address so that an interrupt may branch into one of several interrupt processing routines depending upon which stage the program is in. The ia7601 makes provisions for both generating reset pulses and causing an automatic jump to one of two EROM locations whenever a reset pulse occurs.

The reset circuitry on the ia7601 is triggered in response to a signal applied to the edge connector $\overline{RESIN}$. This is normally a switch closure that momentarily connects $\overline{RESIN}$ to ground. A second option is available to the user in the form of a wire jumper between J1 and J2. As the ia7601 is shipped this jumper is in place; it causes a reset signal to be generated whenever power is applied to the system. Thus a reset will be generated whenever power is applied or the reset switch connected to $\overline{RESIN}$ is closed. If this jumper is removed the power-on reset feature will be defeated.

The output of the reset pulse is also brought to the edge connector for use in the rest of the system. This signal, $\overline{RESOUT}$, is TTL compatible and approximately 3000 ns in length. If the $\overline{RESIN}$ is held low by keeping the switch closed, the $\overline{RESOUT}$ will be stretched accordingly. Since this is an active low output it can be applied directly to most of the latches and flip-flops in the system which generally require active low clears.

Two other inputs are brought to the edge connector and are used to make the system come up in EROM when a reset occurs. If $\overline{ROMRES}$ is tied to $\overline{RESOUT}$, the first instruction executed will be at location 8000H, the address of the first EROM location on the ia7601. At this location the user will place the first instruction of his initialization routine,

generally an LXI SP to set the values of the stack pointer. A second input, MONRES acts like the first except that the first instruction to be executed will be from location 4000H. In practice MONRES is seldom used by itself. The real purpose of this input is to tie it together with ROMRES to RESOUT. This has the effect of causing the first instruction to be executed from location C000H which is the first location in the system monitor.

Thus the two inputs allow the user to pick which of three locations will be the first to be accessed when a reset is applied. The OEM user will generally pick 8000H because it is the first location in the EROM contained on the ia7601. In systems being used in the development mode location C000H will be used since it corresponds to the first instruction of the system monitor. The OEM user will simply hardwire ROMRES to RESOUT; the development system adds logic and switches to the inputs so that the user can reset to either the first instruction of his program or to the system monitor.

RAM AND EROM DISABLES

Although the 8080A drives 16 address lines directly, the ia7601 and expander boards uses 20 address lines. These are decoded by the memory address decoders just as though they are conventional address lines. The lines a re not generated by the ia7601 itself but by the user in the way he connects certain of the edge connector contacts. If EROMDISAB1 is pulled low by tying it to the ground connection, the EROMs on the ia7601 cannot be accessed. If RAMDISAB1 is pulled low, the RAMs on the ia7601 cannot be accessed. Needless to say, if this were the whole story this would not be a very valuable feature. In fact, there are two additional lines present on the system edge connectors. These lines are not used by the ia7601 itself, though they form a very valuable part of the addressing system. The two additional lines are EROMDISAB2 and RAMDISAB2. The expander RAM and EROM boards in the 7600 system decode these lines along with the other conventional address lines. Jumpers on these boards allow the user to configure the boards with EROM as either EROM1 or EROM2. Similarly the RAM boards can be configured as either RAM1 or RAM2.

Assume that the user wishes to debug a program. He adds two ia7621 RAM boards to the ia7601 TCS boards. Both RAM boards are configured as RAM2. By pulling EROMDISAB1 low on the system edge connector, the EROM on the ia7601 will be disabled and out of the system. But if the two RAM boards have been selected by the DIP switches on those boards to match the system memory addresses of the ia7601 EROM, a program can be written into the RAM and executed out of RAM just as though it were in EROM. Since all Iasis memories execute at the same speed, no provision for WAIT states need be made in either EROM or RAM. These provisions allow the user to assemble a ROM simulator with only two standard RAM boards. The RAM and I/O ports of the ia7601 will be completely functional while the program is running out of the RAM boards instead of the on-board EROM.

The RAM and EROM disable lines play an important part in the development system. In that system, the lines are pulled low under control of the user via the front panel. This allows him to quickly run a program out of RAM through the ia7601 under development. It also gives him a great deal of flexibility in the design and debugging operations on custom systems.

I/O PORTS. One of the most impressive features of the Iasis system is in its use of bipolar TTL logic to implement the I/O ports. This is a major departure from other systems which use programmable MOS devices for I/O. These are convenient for the manufacturer but dangerous for the user since they are susceptible to damage and logic errors from noise spikes when these traces are brought off the board. As a result, users have been forced to buffer the inputs and outputs into these devices with TTL to provide noise protection for the MOS. These buffering devices add size and cost to the MOS port and negate its advantages. The Iasis I/O port design sidesteps this problem by using all bipolar circuits in the I/O ports. The latches no longer need to be buffered for protection.

In addition, several wrinkles have been added to the ports. Using the ia7601 TCS board as an example, the board contains six 8-bit ports. Two of these are input ports, two are output ports and two can be programmed by the user under software control to be either input or output. All ports contain latches and full handshaking logic implemented completely in bipolar logic.

Each input port contains two handshaking lines. HSIN is an input to the port that controls the strobing of data into the port. While HSIN is high, the data in the port latches will follow the data input lines. The outputs of the latches are inhibited so they are not "on" the bus. When HSIN is pulled low, the inputs to the

latches are inhibited so the data held in the latches cannot change. $\overline{\text{HSIN}}$ can be thought of as an active low signal that tells the port when the data on its inputs is correct. In some applications the user may not want this strobing action; leaving $\overline{\text{HSIN}}$ disconnected will eliminate the feature and the data in the latches will simply follow the inputs.

Approximately 40 ns after the input port receives the $\overline{\text{HSIN}}$ signal telling it that the data on its inputs is stable, the port responds by pulling its $\overline{\text{HSOUT}}$ output line low. This is signal to the peripheral inputting data to the port that the port has latched the data and has inhibited the inputs. The peripheral can now start assembling the next data byte to input to the port. It must, however, hold the $\overline{\text{HSIN}}$ line low until it receives a signal that the CPU has read the data out of the port. Letting $\overline{\text{HSIN}}$ go back high prior to this signal will destroy the data in the port before it has been read.

When the CPU has read the data out of the port it will signal this fact to the peripheral by letting $\overline{\text{HSOUT}}$ go back high. The rising edge of $\overline{\text{HSOUT}}$ is therefore the signal that the port can be strobed again to latch new data into its inputs.

Each of the ports can be jumpered selected by the user to operate in one of three modes. In the data mode, the port simply retains the data in its latches until read by the CPU. If the user jumper selects the port to be an interrupting port, the $\overline{\text{HSIN}}$ signal that serves to lock the data into the latches will also cause an interrupt to the CPU. Finally if the automatic WAIT is selected instead of the interrupt, the CPU can attempt to read the port before the peripheral has delivered data to the port. If the port is selected by the CPU for a read operation but $\overline{\text{HSIN}}$ is still high meaning the data in the latches is not yet correct, an automatic WAIT signal to the CPU will be generated. The system will idle in this mode until the peripheral signals that the data is ready by pulling $\overline{\text{HSIN}}$ low. The CPU then completes the port read operation and program execution resumes.

The automatic WAIT option is an extremely useful feature for interfacing to other peripherals where there is a speed mismatch. Normally this is done by having the peripheral cause an interrupt to the CPU. The interrupting method has the disadvantage of requiring time to PUSH the various registers onto the stack before the data can be processed. This slows the rate at which data can be transferred into the system. By using the automatic WAIT option, the CPU can sit and wait for the data to become available and then read it instantly without having to go through an interrupt service routine.

Frequently the system will have to read an input port testing for a signal or switch closure. This is generally done with a loop in the program that is exited by a conditional jump. The use of the automatic WAIT option completely removes the need for this portion of the program since the same effect results from connecting the automatic WAIT jumper.

Much of the discussion relative to input ports applies equally to the output ports. Now the $\overline{\text{HSIN}}$ is used to tell the port that the peripheral can accept more output data from the port. This line is not used as often as in the input port since it logically only makes sense when the output port is being used in the automatic WAIT or interrupting mode. In the interrupting mode, $\overline{\text{HSIN}}$ tells the system that it is ready for data. This interrupts the CPU which then processes the data output through the use of an interrupt service routine. Thus the writing of data from the CPU to the output port is done by the interrupt service routine. When $\overline{\text{HSOUT}}$ goes high it is the signal to the peripheral that the data at the output port is correct and the peripheral may process it. If the automatic WAIT is used, the CPU tries to write data to the output port but fails since the peripheral has not signaled the port that it is ready. The CPU enters a WAIT state where it idles until the port receives the ready signal from the peripheral. The port is sent the output data by the CPU which then resumes program execution.

In addition to the two input ports and two output ports, the ia7601 also has two programmable ports. These are also bipolar and have the interrupting and automatic WAIT options of the regular ports. They can be programmed by the user by writing into a special internal latch whether each is to be an input port or an output port. This can be changed during the course of the program so that one of these bidirectional ports can both send and receive data.

TIMING PULSE OUTPUT. One of the more difficult problems with a microcomputer is the seemingly easy task of simply keeping track of time. At first glance this should involve nothing more than incrementing some register at a regular time. Since most systems have some sort of crystal clock this should provide a time base. In fact, the timing signals usually made available to the user are much too short to be read by the system as a timing pulse. The ia7601 has solved this problem by using the 0_2 clock to provide a timing

base to a divide by 20 circuit. The output of this circuit is a 1 ms pulse that is brought to the edge connector. By tying this to one on the input ports or interrupt lines an extremely accurate timing system is produced. These pulses are easily long enough to be read easily or even be serviced by an interrupt handling program.

ON-BOARD RELAY. Frequently the industrial user is faced with the problem of having to interface the system to a signal that may contain very large transient noise spikes. More frequently he will have to face the problem of driving a load that requires a higher voltage or perhaps even 120 VAC. If the requirements are for only one of these loads, the on-board reed relay on the ia7601 board may be the answer.

The relay is a molded DIP reed relay with normally open SPST contacts rated at 10 VA or .500 mA or 150 VDC max. The coil requires a nominal voltage of 5V and 50 mW. Both coil and contacts are brought to the edge connector. A discrete transistor is included on the board in close proximity to the relay so that the user can jumper the transistor into the coil circuit. This allows the user to drive the relay from any of the output ports.

INTERRUPT STRUCTURE. The ia 7601 uses a combination of hardware priority and polled interrupts. All eight of the interrupt lines are brought to the edge connector for the user. However the I/O ports have been dedicated to certain interrupt lines.

$INTRQ_0$ — This is the high priority interrupt and is reserved for the user's power down save routine. If the user is not using the separate power supply line and CMOS RAMs, this interrupt request line is available for other purposes.

$INTRQ_1$ — This is the second priority line and is reserved for the keyboard of the development system. Any key closure on the keyboard will cause a strobe to this interrupt line which then causes the system to enter the monitor routine. If user is not going to use the development system this interrupt becomes available for other purposes.

$INTRQ_2$ — This interrupt is dedicated to bidirectional port #0 on the ia7601 board. As shipped, the port is jumpered connected in the interrupt mode, but if the user disconnects the jumper or elects to use the automatic WAIT option, this interrupt becomes available for other purposes.

$INTRQ_3$ — This interrupt is dedicated to bidirectional port #1 on the ia7601 board. The same comments about $INTRQ_2$ also apply to this interrupt.

$INTRQ_4$
$INTRQ_5$
$INTRQ_6$
$INTRQ_7$ Any of the input or output ports of the entire 7600 system can be jumpered onto either of these four interrupt lines. If only the ia7601 is to be used in the system, this allows the user to have a priority interrupt line dedicated to each of the six ports. If additional I/O ports are to be added in the form of other boards, these ports will share the four Interrupt lines, $INTRQ_4$—$INTRQ_7$. The CPU then polls these I/O ports to see which caused the interrupt.

EXPANDING THE SYSTEM. Because the ia7601 has a set of fully buffered address and data buses as well as a full set of control signals brought to the edge connector, the user can easily expand the system by adding EROM, RAM or I/O boards. In addition, certain dedicated I/O boards like cassette interface, floppy disk interface and CRT interface boards are available. These allow the user to configure either more elaborate OEM systems or development systems. Although the boards have been designed with the industrial OEM user in mind, they are also flexible enough to be capable of making up a wide variety of development systems. Some of these have been outlined in this chapter.

Chapter Eight
Applications

Every microcomputer engineer, when describing the wonder of the microcomputer, has been faced with the same disarming question, "What can you do with them?" The question is a difficult one to answer because the microcomputer is such a versatile beast. In one sense it can be viewed as a replacement part; it replaces lots of random TTL logic. That is not very satisfying, especially to the nontechnical person. "I know, but what can you DO with them?" It's like trying to write an application note a nail. Microcomputers are so basic a building block in digital electronics that the uses should really be left up to the user. Notwithstanding the difficulties, we shall attempt to describe some applications.

TIMEKEEPER FOR LEGAL OFFICES. A common problem faced by many lawyers and other professional people who must interface with a large number of clients is the problem of keeping track of the time spent on each client's problems. This can be very difficult since a phone call may come in while working with one client. The time spent on the phone may run into some time and should be deducted from the client's bill. An even worse problem is if the phone call is from a second client. Now the time spent on the phone must be deducted from the first client and added to the second client. Such interruptions are frequent and cause many unintentional errors in the billings each month. Clearly what is needed is a way to keep track of the time spent on each client, including or should we say, in spite of such interruptions.

One solution is to use an ia7601 board to interface to a keyboard and a set of LEDs. In this case the RAMs should be replaced with the CMOS equivalents and a rechargeable battery back-up used for the memory. This will allow the system to be turned off each night without losing the monthly billing records. The CMOS RAMs draw so little power that the discharge rate of the batteries is negligible and can be easily charged in a few minutes each morning by a very low current rickle charger.

The basic timing function of the system is done by the 8080 itself. Since the ia7601 has a crystal clock and the timing pulses are brought out at 2 ms intervals, it is relatively easy to have the CPU idle in a loop that constantly checks for key closures and for timing pulses. By counting these timing pulses an extremely accurate check is kept of the time. One variation is to use the LEDs to function as a normal clock displaying time of the day. In this case the timing pulse output should be tied to one of the interrupt lines so that each timing pulse interrupts the regular program. This allows the CPU to increment the time counter and change the displays to show the new time (since most timing pulses will not correspond to a change in the seconds display, there will be few display changes).

The regular program loop that the CPU spends its time in is searching for a key closure and enabling the LEDs. This is a very similar problem to the one that faced us in the KBDS program of the system monitor in Chapter 6. The CPU uses an output port to enable the various rows of keys and another to enable the individual segments of the LEDs. This allows the keyboard and LEDs to be scanned/operated simultaneously. The key columns would be read using another input port. One way to do this would be to have the keys NANDed together to produce a strobe when any of the keys are closed. This in turn could cause another interrupt of the system. By using the interrupts in this manner, the program can be shortened slightly and since the interrupts are provided, they can be used to our advantage.

The program operates by displaying the time of the day on the LEDs and scanning for key closures. When the operator begins to spend time with a client he enters the client number on the keyboard. This three digit number appears on the display; the time of day disappears, though the CPU is still keeping track of the time by counting the interruptions due to the timing pulse. The process of entering the client number accesses the RAM memory. One location is dedicated to each client. The client number is, in fact, the address of the RAM location containing his information. By using two locations for each client approximately 500 clients can be serviced, each for up to 65,536 time periods. The time period could consist of every 500 timing pulses so that the client could run up as much as 65,536 seconds of time.

When the operator is interrupted, he punches in the number of the interrupting client. Time ceases to be kept on client #1 and begins on client #2. At the end of the

month each of the RAM locations is accessed and the cumulative time records for each client read out. Billings can then be prepared.

LOGIC TESTER. One of the most common applications for microcomputers is in the field of logic testers. Most of these are constructed in the form of some sort of programmable system, so a microcomputer is an obvious choice. In the case of simpler logic elements like TTL gates, the ia7301 computer in a book is a perfect choice since it has enough I/O capability for most logic elements and an on-board keyboard that can be used to input the testing programs. The process is to hook the various bit outputs of the ia7301 output port up to the inputs of the device under test. The inputs of the ia7301 are connected to the outputs of the device under test. All that remains is for the programmer to write data to the output port to exercise the various logic elements in such a way that the outputs are driven and can be monitored by the ia7301 input port. Even if the number of inputs of the logic element exceeds the number of output lines from the ia7301, this can usually be worked around merely by having each output drive more than one input of the logic device. This is possible because the logic devices that exceed the number of outputs of the ia7301 will generally consist of more than one logic element inside the DIP; for example, a dual JK flip-flop. There is no reason why the output lines from the ia7301 cannot drive both flip-flops at once. Fortunately most devices have more inputs than outputs so there is seldom a problem with the single input port of the ia7301

PROCESS CONTROLLER. This is a natural for a microcomputer. Process control applications tend to become a combination of an elaborate timer plus solenoid drivers. A system for processing color film, for example, can be handled by the ia7601 using its timing pulse output to generate a timing base. The output ports are tied to the lasis triac driver board to provide the drive capability for energizing the chemical solenoid valves. The input ports can then be used for switch and key inputs to control and modify the processing cycle.

Refinements would consist of a thermocouple to control the processing temperatures. The CPU continually minitors the thermocouple outputs and adjusts heater voltages through the triac drivers. Elaborate controllers will also monitor chemical concentrations so that the solutions are being continually replenished as they are used.

MAILING LIST PROCESSOR. One of the major problems facing a small company is the maintenance of the company mailing list. Postal rates dictate that bulk mailings be zip code sorted to facilitate handling within the postal system. It is the sorting problem that represents one of the biggest headaches. The mailing list is frequently kept in the form of typed masters which are then xeroxed onto special mailing label sheets designed to be run through the office copier. Unfortunately there is no easy way to add names to the list at any place but the end. Trying to keep a good sized list in zip order on this kind of a system is nearly impossible.

An ia7601 can be used in a fairly elaborate system consisting of an extra RAM board, a pair of floppy disks, a cassette tape deck, keyboard and video monitor. Since these are all available as standard products designed to be compatible with the ia7601 the hardware aspects of the problem are fairly straight forward. The program is fairly involved.

Names and addresses are entered through the keyboard and checked for accuracy on the video monitor. If the entry is correct the address and name is loaded into RAM memory which is acting as buffer storage. Each time the RAM buffer becomes full, the program stops accepting keyboard inputs and loads the data onto the floppy disk. When the addresses have all been loaded onto the floppy, the process of adding the names to the master list begins. The master mailing list is contained on cassette tape. The tape is read in short sections which are loaded into RAM. The files on the floppy disk are then searched for zip codes that must be merged into the list. This is done in RAM memory with the resultant expanded version of the list loaded onto the second floppy. At some point the list may grow beyond the capacity of the second floppy to handle it all. At this time it will have to be loaded onto a second cassette, either through the first deck onto a new tape, or through a second deck. Labels are generated by printing out through a line printer onto label stock.

ORDER ENTRY TERMINAL. The system just described is also capable of handling the order entry function. In this case the records kept on each customer will now include purchasing information such as open account status, buying history, etc. As new customers place orders, their orders are entered through the keyboard and checked on the video monitor. Invoices and packing lists can be gener-

ated on the line printer at the time the order is entered. At any time, the tapes can be read to generate a mailing list for direct mail pieces or other promotional purposes.

In this very brief survey of some of the possible applications for microcomputer systems, we have tried to illustrate applications of varying complexity. The logic device tester is a very viable project for the hobbyist. The mailing and order entry terminal is an extremely complex project that is best suited for an engineering group with in-house programming capability. Between these two extremes are hundreds of other possibilities, yet to try to describe them is an exercise in futility. Each has its own problems that will become apparent only in the construction and testing of the system. To do anything but mention a few applications and suggest some ways to attack the problems would be like trying to write a book on building a house without going to a construction site. Each building style is different.

Yet we felt that some detailed description of the actual design and problem solving process must be given. This was the reason for chapters five and six wherein we described the design and programming of one of our favorite projects, the ia7301 computer in a book. We feel confident that you too will have your favorite projects and that in the next edition of this book we will be able to include them in new chapters. Good luck!

APPENDIX A

THE HEXADECIMAL NUMBERING SYSTEM

Even a quick perusal of this book will make the reader aware of the fact that many of the numbers used in the text and the programs are stated as a combination of numerals and letters. These numbers use a number system called the hexadecimal number system, and numbers so stated are followed by the letter H to clue the reader to this fact. Thus the number 5 means 5 in the decimal system while 5H means 5 in the hexadecimal number system.

As you are probably aware, all modern computers use the binary number system in which the numbers are made up entirely of zeroes and ones. This is an outcome of a simple economic fact, namely, that it is cheaper to build digital logic circuits that recognize only two states than any other kind. This has brought about a situation in which the computers operate in binary even though the human operator thinks in decimal. Where then did the hexadecimal number system come from?

Computers operate so efficiently in the binary number system that it is advantageous to leave the numbers that make up our problems in that number system as much as possible. Only when data is to be outputted to human beings will we have the computer perform a number conversion to convert the binary results into decimal answers that can be understood by humans. Of course, the human operators could perform the conversion back and forth using paper and pencil, but the process is long and prone to errors. The fact that during the vast majority of the time the computer is thinking in binary means that the human programmer and designer must think in binary as well. This is very awkward since with only two symbols, 0 and 1, it takes a lot of 0's and 1's to express even simple decimal numbers. Trying to rattle off a string of 1's and 0's to a fellow programmer can become a mouthful and trying to remember them can be worse. That's where hexadecimal comes in. The conversion from binary to hexadecimal and vice-versa can be done in your head. Since hexadecimal numbers are much shorter than binary or even decimal numbers, they're easy to remember, talk about or work with. Keep in mind, though, that even though the programmer is talking about hexadecimal numbers, the computer itself is still using the 0's and 1's that make up the binary number system. In fact, it is helpful to think of the hexadecimal number system as a shorthand version of binary.

The fact that hexadecimal is easy to convert to binary and back again is due to the fact that it uses exactly 16 symbols, an even power of two whereas decimal uses ten symbols which is not a power of two. The 16 symbols are 0–9 and A–F. Each symbol can replace exactly four binary bits or 1's and 0's. We can make up a conversion table as follows:

Hexadecimal	Binary	Decimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
A	1010	10
B	1011	11
C	1100	12
D	1101	13
E	1110	14
F	1111	15

You will notice that all 16 possible combinations of four binary bits, 1's and 0's, can be represented by one of the 16 hexadecimal numerals. This one to one correspondence is what makes hexadecimal 1 so easy to use. It is only necessary to group the binary number to be converted into groups of fours, starting from the right, and using zeroes to fill in the open group on the left. The 4-bit groups can then be treated like individual 4-bit numbers to make the conversion. Thus 11010101101110110010101000111001O1, a formidable number, is seen to be the equivalent of 356ECA8E5 in hexadecimal by the process below.

0011 0101 0110 1110 1100 1010 1000 1110 0101

3 5 6 E C A 8 E 5

Quite obviously, this is a much easier number to handle for a human than the binary version.

The opposite conversion is just as easy. It is only necessary to substitute the correct four binary bits out of the conversion table for each of the hexadecimal numerals.

APPENDIX B
8080 ARCHITECTURE

This book is based on the assumption that the reader is already somewhat familiar with the hardware aspects of the 8080 microprocessor. If not, this section may be helpful.

The 8080 can be thought of as the computer system minus memory and I/O. Most of the control circuits and all of the manipulative circuits are contained within this single IC. When it is combined with the memory and I/O circuits, a complete computer system results. The only other circuitry needed is a clock generator to provide the system timing base and some driver circuits to increase the fan out of the 8080.

Inside the 8080 itself are a set of specialized registers and the timing and control circuits that define the basic operation of the computer. These timing and control circuits are in turn controlled by the external clock generator and some user interface circuits like RESET. When power is first applied to the system or an external RESET switch closed, a RESET signal is applied to one of the inputs of the 8080. This initializes one of the special registers called the program counter so that it contains all zeroes. The program counter is a 16-bit register and is used to address the memory. Its contents represent the address of the memory location being accessed for program instructions. After each instruction is executed, the program counter is incremented by one.

Programs are written and entered into the system by storing them in consecutive memory locations. For the purpose of this discussion we will assume that the program is written with the first instruction stored in the first location in memory, 0000H. Since the internal program counter is initialized at 0000H by the RESET signal, this will be the memory location accessed for the first instruction fetch. The 8080 fetches instructions and data by sending out an address on a 16-line dedicated bus called the address bus. This bus goes to all of the memory elements and I/O ports. In addition to the address bus is a control bus generally made up of four or more lines that provides the control signals to the memory and external circuits. In the case of this first instruction fetch, the contents of the program counter, 0000H, are placed on the address bus along with a signal on the control bus telling the memory that a memory read operation is being conducted.

When the memory receives the memory read signal on the control bus it interprets the data on the address bus as an address and uses it to fetch the contents of the specified memory location. This data is placed on a third bus that also interconnects the 8080 with the memory and I/O ports. The third bus, called the data bus, is only eight lines wide and is used for carrying data between the various parts of the system. In this case, the memory places the contents of the memory location 0000H, which is the first instruction of the program being executed, onto the data bus where it is picked off by the 8080 and placed in a special internal register called the instruction register. Here it will remain while the control circuits of the 8080 interpret it and execute the instruction. When execution is complete, the program counter is incremented, its contents sent out on the address bus and the contents of the next memory location fetched to the instruction register. In this way the program is fetched from memory and executed, instruction by instruction.

After each instruction is loaded into the instruction register it must be executed. This is done by the control circuits inside the 8080. They use the instruction as an address to a special ROM section also internal to the 8080. Here the instruction is "looked up" in the ROM and control signals generated to cause proper execution. Each instruction will cause a unique set of control signals to be generated; each instruction is therefore executed in a different way. Some instructions are executed in a way that is completely internal to the 8080; others involve other memory locations and I/O ports.

In addition to the instruction regsiter the 8080 contains other registers that can be used by the programmer. Some of these are general purpose; others are specialized and best suited for certain operations. The most important of these is the accumulator. This 8-bit register is used in all of the arithmetic operations of the computer. It can be used for addition and subtraction operations. Rotate left and right operations can be used to build multiplication and division routines. In addition to the arithmetic operations it can also do logical operations like AND, OR and EXCLUSIVE OR.

Most of these operations require two bytes of data or operators. One is the number in the accumulator itself; the second can be provided with the instruction or contained in one of the other 8080 registers. In any case, the result of operating on the two numbers is generally stored in the accumulator so that it

replaces one of the original bytes of data. Of course one of the operators may be contained in the system memory but the process of fetching the data into the 8080 complicates the operation to the point where it is inconvenient to operate on data there unless absolutely necessary. The general-purpose registers on the 8080 itself are ideal for this purpose since operations with the accumulator and any of these registers can be specified with a simple one-byte instruction. For this reason the general-purpose registers can be thought of as "scratch-pad" registers for storing data temporarily while operations manipulating the data take place in the accumulator. Only when these operations are completed will the data be stored back in system memory to free up the general-purpose registers for other operations. This reduces the process of fetching and replacing data in the memory to an absolute minimum which in turn speeds execution of the program and reduces the amount of memory required to store the program.

There are six of these general-purpose registers, B, C, D, E, H and L, in addition to the accumulator, generally labeled A. The H and L registers have a special feature in that they can be used for memory references, either to read or to write. When used in this fashion the contents of the H and L registers are sent out on the address bus and used to send or fetch data back to the 8080. In this case the data is not stored in the instruction register but is sent or received by one of the other general-purpose registers. Used in this way the H/L register pair is said to be operating as a memory pointer. In certain very restricted situations the B,C pair or the D,E pair may operate as a memory pointer also. The accumulator, or A register, is thus used for arithmetic or logical operations while B, C, D, E, L and H are used for temporary scratch-pad applications. H and L can also be used as a memory pointer.

There are many times in the course of a program when a certain group of instructions will be repeated once or even many times. The program can be shortened considerably by treating these instructions as a subroutine. To do this a special instruction labeled a CALL is used. This instruction includes a 16-bit address (the instruction requires three bytes to specify). When the CALL is executed the 16-bit address is loaded directly into the program counter, displacing the previous contents. This causes an immediate jump to the new address which is the address of the first instruction of the subroutine. When the subroutine has been completed it will be necessary to return program execution to the original program. This means that the original contents of the program counter will have to be reloaded so that execution can pick up where it left off. Obviously this return address had to be saved when the subroutine was called.

The saving of return addresses is done right in the system memory. A portion of the memory is blocked off and reserved for the exclusive use of the stack; it is called the stack memory. A special register in the 8080 called the stack pointer is set up to contain the address of the top of the stack memory. Whenever a CALL to a subroutine is executed the contents of the program counter are PUSHed onto the stack meaning they are loaded into two bytes of the stack memory at the location specified by the stack pointer. The stack pointer is then decremented by two so that it points to new, unused memory locations. The last instruction of every subroutine must be a RETurn which causes the stack pointer to be incremented twice so that it points back to the original return address. This address is then reloaded into the program counter so that execution will continue where it left off in the original program.

There is no reason why one subroutine cannot call another so that the subroutines become nested. This is, in fact, good programming practice; practical programs may nest subroutines many levels deep. The use of the stack pointer to keep the return addresses on the stack memory keeps everything straight. Every time a subroutine is called the program counter is pushed onto the stack and the stack pointer decremented by two. As the program execution goes into deeper levels of subroutine nesting, the stack grows downward. As execution comes back out of the subroutine nests the stack pointer is incremented and the return addresses reloaded into the program counter; the stack shrinks upward.

Subroutines used in this fashion lead to a form of programming where the task is broken up into many small units, each handled by a subroutine. These are called by a higher level subroutine, and so on. The small subroutines are program modules that can be combined to form larger and larger programming units.

So far we have been discussing the 8080 as it operates in a free-running mode, going from subroutine to program segment, executing them all in exactly the order they were set down. It is a very desirable feature to be able to modify this order even as the program is being executed. This would allow the system

to treat jobs on a priority basis, changing the program depending on which tasks and data are presented to it. This capability exists in the 8080 and is called the interrupt capability.

A special input on the 8080 called the Interrupt Input is used to accept signals from the user. When an interrupt is received, it is treated much like a subroutine call. The contents of the program counter are pushed onto the stack memory and the stack pointer decremented twice. The interrupt does not in itself provide a jump to address the way the CALL did. After all, interrupt is an input signal and not an instruction. It is up to the system hardware design to electrically jam the jump address onto the address bus at just the right time during the execution cycle. Normally this would take three cycles, one for the jump instruction and two for the jump address. The 8080 has made this process simpler by providing a set of REStart instructions. These are single byte instructions that provide both the jump and the jump address. Of course with only one byte to work with a very limited number of jump addresses can be provided. The most popular of these is RES 7 which is executed if the data bus is jammed with all logic ones. When this happens during an interrupt the 8080 jumps to location 0038H. That location should be loaded with another jump, this time to the exact address of the first instruction of the interrupt processing program segment. When the interrupt has been processed, the program segment is concluded with a RET instruction which causes program execution to return to the original program.

In addition to the registers in the 8080 there are a set of five status flip-flops or flags. These are set or reset during certain instructions depending upon the effect the instruction had on the contents of the accumulator. These include the carry flag, the zero flag, the parity flag, the sign flag and the auxiliary carry flag. If, for instance, decrementing the accumulator leaves it with all zeroes, the zero flag will be set to logic one. There are several instructions within the 8080 instruction set that are conditional depending upon the state of these flag bits. JZ will cause a jump to a specified address ONLY if the zero flag has been set.

The programmer must be very careful to be sure that his program is setting and resetting these flag bits in a way that allows them to be used by the conditional instructions. If he wishes to cause a jump when the contents of the H and L register pair are zero, he cannot use the DCX instruction which treats the registers as a 16-bit register and decrements it by one. The DCX instruction does not set the zero flag so that any conditional instruction following the DCX will not react to the fact that the pair may contain zero contents. Instead a DCR instruction will have to be used to decrement the L register. If that operation produces a borrow (sets the carry flag) a DCR will have to be executed to the H register. If that causes the zero flag to be set, then both H and L must then contain zero.

APPENDIX C
THE 8080 INSTRUCTION SET

This section of the book will describe the instruction set and programming practices for the 8080. It is based on the Intel assembler language and uses the mnemonics and syntax of that programming language. This is consistant with the programming examples used throughout this text.

Assembly language instructions must follow a preestablished format exactly. This format defines four fields or spaces within each instruction line. Some of these are optional, others are used for certain instructions and not others, and some must be used for every instruction. Whenever a field is used, however, the entries in that field must follow the rules set down for that field.

Field number one is called the label field. This can be thought of as the name of the program step. In higher level languages this field is normally used for the line numbers of the various instructions. In assembly language programming for microcomputers, however, the lines are identified with alphanumeric names or labels containing no more than 5 characters. The label must begin with a letter of the alphabet or one of the characters @ or ? It is important to realize that not every line must contain a label. Only those instructions that are used as the destination of other jump instructions or subroutine calls need labels, and since the labels use up the memory capacity of the development system rather quickly when assembling the program the labels should be kept to the minimum needed.

Use of the label field allows the programmer to specify jump addresses without knowing the actual address assigned to any particular location. Thus the instruction JMP THERE will cause a jump to the instruction that carries the label THERE. The development system must be able to identify labels when it scans the instruction line; this is done by adding a colon to the end of every label.

```
HERE:   MVI C,0
```

FIG. B-1. Use of the colon after labels.

Note in Fig. B-1 that the colon appears after the label HERE when it is used as a label, but not after HERE when it is used as a memory reference in the jump instruction.

The second field is the code field and is used to specify the instruction. This must be one of the recognized mnemonics in the assembler language. Thus JMP is a legitimate instruction mnemonic that can be used in the code field; JUMP is not. All of the mnemonics in the 8080 instruction set are summarized in this section. Obviously every instruction must have an entry in the code field.

Many instructions require data or addresses as part of the instruction. Move Immediate, for instance, requires both a register identifier and the byte of data to be loaded into the register. MVI A, 0CH indicates that the hexadecimal byte of data 0CH is to be loaded into the accumulator. The letter A and the 0CH are called operands; they specify data, registers or addresses that the computer will need to execute the instruction. Some instructions do not need other data to be executed. They will not make use of this third field which is termed the operand field.

The first three fields are all that are necessary to write or execute a program in assembly language. Good programmers, however, will make extensive use of a fourth field called the comment field. This is completely ignored by the computer but provides space for the programmer to insert comments about the workings of the program. These make it much simpler for someone else to troubleshoot or use the program. Naturally we won't want the computer to be confused by these comments so we always precede them by a semicolon on each line. As soon as the computer sees a semicolon it ignores everything to the right of it and goes on to the next line.

CMC — Complement Carry

Mnemonic/Code	CMC
Operand	—
Flags Affected	C

This instruction causes the carry flip-flop to be complemented. If the carry flag is 0, CMC will set it to 1, If the carry flag is 1, CMC will reset it to 0.

STC — Set Carry

Mnemonic/Code	STC
Operand	—
Flags Affected	C

This instruction sets the carry flip-flop to 1. If the carry flip-flop is already 1, STC will have no affect.

INR — Increment Register or Memory

Mnemonic/Code	INR
Operand	R
Flags Affected	Z, S, P, AC

where R is either A, B, C, D, E, H, L or M.

This instruction causes the contents of the specified register, R, to be incremented by one.

DCR — Decrement Register or Memory

Mnemonic/Code	DCR
Operand	R
Flags Affected	Z, S, P, AC

where R is either A, B, C, D, E, H, L or M.

This instruction causes the contents of the specified register, R, to be decremented by one.

CMA — Complement Accumulator

Mnemonic/Code	CMA
Operand	—
Flags Affected	None

The contents of the accumulator are complemented using the one's complementing process.

DAA — Decimal Adjust Accumulator

Mnemonic/Code	DAA
Operand	—
Flags Affected	Z, S, P, C, AC

The contents of the accumulator are modified to form two 4-bit BCD numbers. This is done by examining the least four significant bits have been modified, the four MSB are considered. Notice that if the first part of the DAA process caused the addition of six to the accumulator, the four MSB will have been changed. This makes it very important to consider the four LSB first. If the four MSB now represent a number that is greater than 9 or if the regular carry flag has been set to 1, the DAA instruction will cause 6 to be added to the four MSB.

Thus the four LSB are treated as a BCD number and adjusted accordingly. The result of the adjustment may be to cause the Auxilliary Carry flag to be set. The four MSB are than treated as a BCD number. During the modification of the MSB the Auxilliary Carry will be examined. If six added to the four MSB causes a carry, the carry flag will be set.

NOP — No Operation

Mnemonic/Code	NOP
Operand	—
Flags Affected	None

No operation occurs. The program counter is incremented by one and execution continues at the next instruction. This instruction is useful for adding delays to timing loops to make up exact time delays. It is also useful in writing a program for the first time to sprinkle a few NOPs through each of the subroutines so that instructions can be added later without having to reassemble. Then on the last assembly pass the extra NOPs can be removed.

MOV — Move

Mnemonic/Code	MOV
Operand	R, S
Flags Affected	None

where R and S are either A, B, C, D, E, L, H or M.

This instruction causes one byte of data to be moved from the register designated by S to the register designated by R. If M is specified as one of the registers, the contents of the memory location addressed by L and H will be used for M. MOV A,A and other variations where R and S are designated to be the same register are valid instructions but will cause no operation to be performed; they are the equivalent of NOPs.

STAX — Store Accumulator

Mnemonic/Code	STAX
Operand	RP
Flags Affected	None

where RP is one of the register pairs denoted by B (B/C) or D (D/E).

This instruction causes the contents of the accumulator to be stored in the memory location addressed by the M register pair RP.

LDAX – Load Accumulator

Mnemonic/Code	LDAX
Operand	RP
Flags Affected	None

where RP is one of the register pairs denoted by B (B/C) or D (D/E).

This instruction causes the contents of the memory location addressed by the register pair RP to be loaded into the accumulator.

ADD – Add Register or Memory to Accumulator

Mnemonic/Code	ADD
Operand	R
Flags Affected	C, S, Z, P, AC

where R is either A, B, C, D, E, L, H or M.

This instruction causes the contents of the specified register to be added to the accumulator with the results stored in the accumulator. A carry from the addition will be reflected in the status of the carry flag. If M is specified as the register, the contents of the memory location addressed by L and H will be used.

ADC – Add Register or Memory to Accumulator with Carry

Mnemonic/Code	ADC
Operand	R
Flags Affected	C, S, Z, P, AC

where R is either A, B, C, D, E, L, H or M.

This instruction causes the contents of the specified register plus the carry bit to be added to the contents of the accumulator. In this operation the carry bit is treated as a LSB. The results are stored in the accumulator. If a carry out is produced as a result of the addition operation, the carry flag will be set to reflect this fact.

SUB – Subtract Register or Memory from Accumulator

Mnemonic/Code	SUB
Operand	R
Flags Affected	C, S, Z, P, AC

where R is either A, B, C, D, E, L, H or M.

This instruction causes the specified register to be subtracted from the contents of the accumulator using two's complement arithmetic and the result stored inthe accumulator. If there is no borrow out of the high order bit position, the carry flag will be set to 1. If a borrow occurs, the carry flag will be reset to 0. If M is specified as the register, the contents of the location addressed by H and L will be used.

SBB – Subtract Register or Memory from Accumulator with Borrow

Mnemonic/Code	SBB
Operand	R
Flags Affected	C, S, Z, P, AC

where R is either A, B, C, D, E, L, H or M.

This instruction causes the contents of the carry bit flip-flop to be added to the contents of the specified register. This register is then subtracted from the accumulator using two's complement arithmetic and the result stored in the accumulator. If M is specified as the register, the contents of the location addressed by H and L will be used.

ANA – Logically AND Register or Memory with Accumulator

Mnemonic/Code	ANA
Operand	R
Flags Affected	C, Z, S, P

Where R is either A, B, C, D, E, L, H or M.

This instruction causes the contents of the specified register to be logically ANDed, bit by bit, with the contents of the accumulator and the result stored in the accumulator. The carry flag is reset to 0. If M is specified, the contents of the memory location addressed by H and L will be used.

XRA – Logically EXCLUSIVE-OR Register or Memory with Accumulator

Mnemonic/Code	XRA
Operand	R
Flags Affected	C, Z, S P

where R is either A, B, C, D, E, L, H or M.

This instruction causes the contents of the specified register to be EXCLUSIVE-ORed , bit by bit, with the contents of the accumulator and the result stored in the accumulator. The carry flag is reset to 0. If M is specified, the contents of the memory location addressed by H and L will be used.

ORA – Logically OR Register or Memory with Accumulator

Mnemonic/Code	ORA
Operand	R
Flags Affected	C, Z, S, P

where R is either A, B, C, D, E, L, H or M.

This instruction causes the contents of the specified register to be ORed, bit by bit, with the contents of the accumulator and the result stored in the accumulator. The carry flag is reset to 0. If M is specified the contents of the memory location addressed by H and L will be used.

CMP – Compare Register or Memory with Accumulator

Mnemonic/Code	CMP
Operand	R
Flags Affected	C, Z, S, P, AC

where R is either A, B, C, D, E, L, H or M.

The contents of the specified register are compared to the contents of the accumulator and the zero and carry flags set according to the result of the comparison. The contents of the accumulator are unchanged by the operation. If the two quantities are equal, the zero bit is set to 1. If the specified register is larger than the accumulator, the carry bit will be set to 1; if it is smaller the carry will be reset to 0.

RLC – Rotate Accumulator Left

Mnemonic/Code	RLC
Operand	–
Flags Affected	C

This instruction causes the contents of the accumulator to be shifted one bit position to the left. The high order bit is shifted around to the lower order bit position. In addition, the high order bit is duplicated in the carry bit.

RRC – Rotate Accumulator Right

Mnemonic/Code	RRC
Operand	–
Flags Affected	C

This instruction causes the contents of the accumulator to be shifted one bit position to the right. The low order bit is shifted around to the high order bit position. In addition, the low order bit is duplicated in the carry bit.

RAL – Rotate Accumulator Left through Carry

Mnemonic/Code	RAL
Operand	–
Flags Affected	C

This instruction causes the contents of the accumulator to be rotated one bit position to the left. The carry bit is treated by this instruction as a ninth bit position, that is the high order bit of the accumulator is shifted into the carry bit and the prior contents of the carry bit is shifted into the low order bit position.

RAR – Rotate Accumulator Right through Carry

Mnemonic/Code	RAR
Operand	–
Flags Affected	C

This instruction causes the contents of the accumulator to be rotated one bit position to the right. The carry bit is treated by this instruction as a ninth bit position, that is the low order bit of the accumulator is shifted into the carry bit and the prior contents of the carry bit is shifted into the high order bit position.

PUSH – Push Data Onto the Stack

Mnemonic/Code	PUSH
Operand	RP
Flags Affected	None

where RP is either B (for registers B and C), D (for registers D and E), H (for registers H and L) or PSW (for the flags and accumulator).

This instruction causes the specified register pair to be saved in the stack memory and the stack pointer decremented by two. The first register of the pair (the one used as the operand) is saved at the location one less than the stack pointer. The second register is saved at the location two less than the stack pointer. Notice that the stack pointer is decremented first and the register then saved. This means that a PUSH B instruction will not save the B register at the location addressed by the stack pointer but at the location one less than the stack pointer. The C register will then be saved at two less than the stack pointer.

When PUSH PSW is used to save the flags at one less than the stack pointer and the accumulator at two less than the stack pointer, the flags are saved as follows:

Bit Position No.

7 – state of sign flag
6 – state of zero flag
5 – always 0
4 – state of auxilliary carry
3 – always 0
2 – state pf parity flag
1 – always 1
0 – state of carry bit

POP — Pop Data off Stack

Mnemonic/Code	POP
Operand	RP
Flags Affected	None

where RP is either B (for registers B and C), D (for registers D and E), H (for registers H and L) or PSW (for the flags and accumulator).

This instruction causes the specified register pair to be retrieved from the stack memory and the stack pointer incremented by two. The memory location indicated by the stack pointer is loaded into the second register of the pair. The memory location one greater than that indicated by the stack pointer is loaded into the first register of the pair.

DAD — Double Add

Mnemonic/Code	DAD
Operand	RP
Flags Affected	C

where RP is either B (for registers B and C), D (for registers D and E), H (for registers H and L) or SP (for the stack pointer).

The 16-bit contents of the specified register pair are added to the 16-bit contents of the H and L registers and the result stored in H and L. If the register pair specified is H, the effect will be to double the contents of H and L.

INX — Increment Register Pair

Mnemonic/Code	INX
Operand	RP
Flags Affected	None

where RP is either B (for registers B and C), D (for registers D and E), H (for registers H and L) and SP for the stack pointer.

The 16-bit contents of the specified register pair are incremented by 1.

DCX — Decrement Register Pair

Mnemonic/Code	DCX
Operand	RP
Flags Affected	None

where RP is either B (for registers B and C), D (for regirsters D and E), H (for registers H and L) and SP for the stack pointer.

The 16-bit contents of the specified registers pair are decremented by 1.

XCHG — Exchange Registers

Mnemonic/Code	XCHG
Operand	—
Flags Affected	None

The 16-bit contents of the D and E registers are exchanged with the contents of the H and L registers. The exchange is between D and H, and E and L.

XTHL — Exchange Stack

Mnemonic/Code	XTHL
Operand	—
Flags Affected	None

The contents of the L register are exchanged with the contents of the memory location addressed by the stack pointer. The contents of the H register are exchanged with the contents of the memory location one greater than that addressed by the stack pointer.

SPHL — Load SP from H and L

Mnemonic/Code	SPHL
Operand	—
Flags Affected	None

The contents of the H and L registers are loaded into the stack pointer. The prior contents of the stack pointer are lost; H and L remain unchanged.

LXI — Load Register Pair Immediate

Mnemonic/Code	LXI
Operand	RP, D16
Flags Affected	None

where RP is either B (for registers B and C), D (for registers D and E), H (for registers H and L) and SP for the stack pointer.

This instruction causes the register pair specified to be loaded with the sixteen bits of data specified by D16.

MVI — Move Immediate Data

Mnemonic/Code	MVI
Operand	R, D8
Flags Affected	None

where R is either A, B, C, D, E, L, H or M. D8 represents 8 bits of data.

This instruction causes the 8 bits of data specified by D8 to be loaded into the register specified by R. If M is specified by H and L are used.

ADI — Add Immediate to Accumulator

Mnemonic/Code	ADI
Operand	D8
Flags Affected	C, S, Z, P, AC

This instruction causes the 8-bits of data represented by D8 to be added to the contents of the accumulator using two's complement arithmetic and the result stored in the accumulator.

ACI – Add Immediate to Accumulator with Carry

Mnemonic/Code	ACI
Operand	D8
Flags Affected	C, S, Z, P, AC

This instruction causes the 8-bits of data represented by D8 plus the contents of the carry flip-flop to be added to the contents of the accumulator using two's complement arithmetic and the result stored in the accumulator. The carry bit is treated as the most LSB.

SUI – Subtract Immediate from Accumulator

Mnemonic/Code	SUI
Operand	D8
Flags Affected	C, S, Z, P, AC

This instruction causes the 8-bits of data represented by D8 to be subtracted from the contents of the accumulator using two's complement arithmetic and the result stored in the accumulator. If a borrow out of the high order bit position occurs the carry bit is set. If no borrow occurs, the carry is reset to 0.

SBI – Subtract Immediate from Accumulator with Borrow

Mnemonic/Code	SBI
Operand	D8
Flags Affected	C, S, Z, P, AC

This instruction causes the carry bit to be added to the 8-bits of data represented by D8. This value is then subtracted from the accumulator using two's complement arithmetic. The result is stored in the accumulator. If a borrow out of the high order bit position occurs the carry bit is set. If no borrow occurs, the carry is reset to 0.

ANI – AND Immediate with Accumulator

Mnemonic/Code	ANI
Operand	D8
Flags Affected	C, Z, S, AC

This instruction causes the 8-bits of data represented by D8 to be logically ANDed, bit by bit, with the contents of the accumulator and the results stored in the accumulator. The carry flag is reset to 0.

XRI – EXCLUSIVE-OR Immediate with Accumulator

Mnemonic/Code	XRI
Operand	D8
Flags Affected	C, S, Z, P

This instruction causes the 8-bits of data represented by D8 to be logically EXCLUSIVE-ORed, bit by bit, with the contents of the accumulator and the results stored in the accumulator. The carry flag is reset to 0.

ORI – OR Immediate with Accumulator

Mnemonic/Codw	ORI
Operand	D8
Flags Affected	C, Z, S, P

This instruction causes the 8-bits of data represented by D8 to be logically ORed, bit by bit, with the contents of the accumulator and the results stored in accumulator. The carry flag is reset to 0.

CPI – Compare Immediate with Accumulator

Mnemonic/Code	CPI
Operand	D8
Flags Affected	C, Z, S, P, AC

This instruction causes the 8-bits of data represented by D8 to be compared with the contents of the accumulator and the zero and carry flags set according to the result of the comparison. The contents of the comparison are unchanged by the operation. If the two quantities are equal, the zero bit is set to 1. If the immediate data in D8 is larger than the contents of the accumulator, the carry bit will be set to 1; if D8 is smaller the carry will be reset to 0.

STA – Store Accumulator Direct

Mnemonic/Code	STA
Operand	D16
Flags Affected	None

This instruction causes the contents of the accumulator to be stored at the memory location specified by the address D16.

LDA – Load Accumulator Direct

Mnemonic/Code	LDA
Operand	D16
Flags Affected	None

This instruction causes the accumulator to be loaded with the contents of the memory location specified by the address D16.

SHLD – Store H and L Direct

Mnemonic/Code	SHLD
Operand	D16
Flags Affected	None

This instruction causes the contents of the L register to be stored in the memory location specified by D16. The H register is stored at the location one larger than D16.

LHLD – Load H and L Direct

Mnemonic/Code	LHLD
Operand	D16
Flags Affected	None

This instruction causes the contents of the memory location specified by D16 to be stored in the L register. The contents of the location one largser than D16 is stored in the H register.

PCHL – Load Program Counter

Mnemonic/Code	PCHL
Operand	–
Flags Affected	None

This instruction causes the contents of the H and L registers to be loaded into the program counter. This causes the program execution to be continued at the instruction specified by H and L.

JMP – Jump

Mnemonic/Code	JMP
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16.

JC – Jump if Carry

Mnemonic/Code	JC
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the carry flag is set to 1. If the carry flag is 0, the JC instruction is executed as a NOP.

JNC – Jump if No Carry

Mnemonic/Code	JNC
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the carry flag is set to 0. If the carry flag is 1, the JNC is executed as a NOP.

JZ – Jump if Zero

Mnemonic/Code	JZ
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the zero flag is set to 1. If the zero flag is set to 0, the JZ is executed as a NOP.

JNZ – Jump if Not Zero

Mnemonic/Code	JNZ
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the zero flag is set to 0. If the zero flag is set to 1, the JNZ is executed as a NOP.

JM – Jump if Minus

Mnemonic/Code	JM
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the sign bit is set to 1. If the sign bit is reset to 0, the JM is executed as a NOP.

JP – Jump if Positive

Mnemonic/Code	JP
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the sign bit is 0. If the sign bit is 1, the JP is treated as a NOP.

JPE – Jump if Parity Even

Mnemonic/Code	JPE
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the parity flag is set to 1 indicating even parity. If the parity flag is reset to 0 the JPE will be treated as a NOP.

JPO – Jump if Parity Odd

Mnemonic/Code	JPO
Operand	D16
Flags Affected	None

This instruction causes the program execution to continue at the memory location specified by D16 if the parity flag is reset to 0 indicating odd parity. If the parity flag is set to 1 the JPO will be treated as a NOP.

CALL — Call Subroutine

Mnemonic/Code	CALL
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16. The return address is saved in the stack memory and the stack pointer decremented by two.

CC — Call if Carry

Mnemonic/Code	CC
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the carry flag is set to 1. If the carry flag is reset to 0, the CC will be treated as a NOP.

CNC — Call if No Carry

Mnemonic/Code	CNC
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 is the carry flag is reset to 0. If the carry flag is set to 1, the CNC will be treated as a NOP.

CZ — Call if Zero

Mnemonic/Code	CZ
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the zero flag is set to 1. If the zero flag is reset to 0, the CZ will be treated as a NOP.

CNZ — Call if not ZERO

Mnemonic/Code	CNZ
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the zero flag is reset to 0. If the flag is set to 0, the CNZ will be treated as a NOP.

CM — Call if Minus

Mnemonic/Code	CM
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the sign bit is set to 1. If the sign bit is reset to 0, the CM will be treated as a NOP.

CP — Call if Positive

Mnemonic/Code	CP
Operand	D16
FLags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the sign bit is reset to 0. If the sign bit is set to 1, the CP will be treated as a NOP.

CPE — if Parity Even

Mnemonic/Code	CPE
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the parity bit is set to 1. If the parity bit is reset to 0, the CPE will be treated as a NOP.

CPO — Call if Parity Odd

Mnemonic/Code	CPO
Operand	D16
Flags Affected	None

This instruction causes program execution to continue at the address specified by D16 if the parity bit is reset to 0. If the parity bit is set to 1, the CPO will be treated as a NOP.

RET — Return

Mnemonic/Code	RET
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer incremented by two. Program execution continues at this address.

RC — Return if Carry

Mnemonic/Code	RC
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer incremented by two IF the carry flag is set to one. Program execution will continue at this address. If the carry bit is reset to 0, the RC will be treated as a NOP.

RNC — Return if No Carry

Mnemonic/Code	RNC
Operand	---
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by two IF the carry bit is reset to 0. If the carry bit is set to 1, the RNC will be treated as a NOP.

RZ — Return if Zero

Mnemonic/Code	RZ
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by two IF the zero flag is set to 1. If the zero flag is reset to 0, the RZ will be treated as a NOP.

RNZ — Return if Not Zero

Mnemonic/Code	RNZ
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by two IF the zero flag is reset to 0. If the zero flag is set to 1, the RNZ will be treated as a NOP.

RM — Return if Minus

Mnemonic/Code	RM
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by two IF the sign bit is set to 1. If the sign bit is reset to 0, the RM will be treated as a NOP.

RP — Return if Positive

Mnemonic/Code	RP
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by two IF the sign bit is reset to 0. If the sign bit is set to 1, the RP will be treated as a NOP.

RPE — Return if Parity Even

Mnemonic/Code	RPE
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by two IF the parity flag is set to 1. If the parity bit is reset to 0, the RPE will be treated as a NOP.

RPO — Return if Parity Odd

Mnemonic/Code	RPO
Operand	—
Flags Affected	None

This instruction causes an address to be POPed off the stack memory and the stack pointer to be incremented by ?

RST — Restart

Mnemonic/Code	RST
Operand	N
Flags Affected	None

where N is an integer in the range 0 — 7.

This instruction is seldom used in the program listing itself. Instead it is created by the hardware and jammed onto the data bus when the 8080 acknowledges an interrupt. It thus over rides any regular instruction that would normally be on the bus and causes program execuation to continue at an address created by evaluating 11EXP111 where EXP is the three bit binary equivalent of N. The current conents of the program counter are pushed onto the stack memory to provide a return address. This instruction allows eight interrupting devices to each place a separate RST instruction on the bus and hence, to access a separate and distinct interrupt service routine.

EI — Enable Interrupts

Mnemonic/Code	EI
Operand	—
Flags Affected	None

This instruction sets the interrupt enable flip-flop in the 8080 so that incoming interrupts will be recognized and processed by the system.

DI — Disable Interrupts

Mnemonic/Code	DI
Operand	—
Flags Affected	None

This instruction resets the interrupt enable flip-flop in the 8080 so that incoming interrupts will be ignored. the flip flop is automatically reset when the 8080 is first energized so that interrupts cannot be serviced when the system is first powered up. They are also disabled when an interrupt is acknowledged. This prevents the same interrupt from continually causing an interrupt and in fact, interrupting itself.

IN — Input

Mnemonic/Code	IN
Operand	D8
Flags Affected	None

where D8 represents the address of the input port.

This instruction causes an 8-bit byte of data to be read from the input port specified by D8. The data is read into the accumulator.

OUT – Output

Mnemonic/Code	OUT
Operand	D8
Flags Affected	None

where D8 represents the address of the output port.

This instruction causes the contents of the accumulator to be sent out on the data bus to the output port specified by D8.

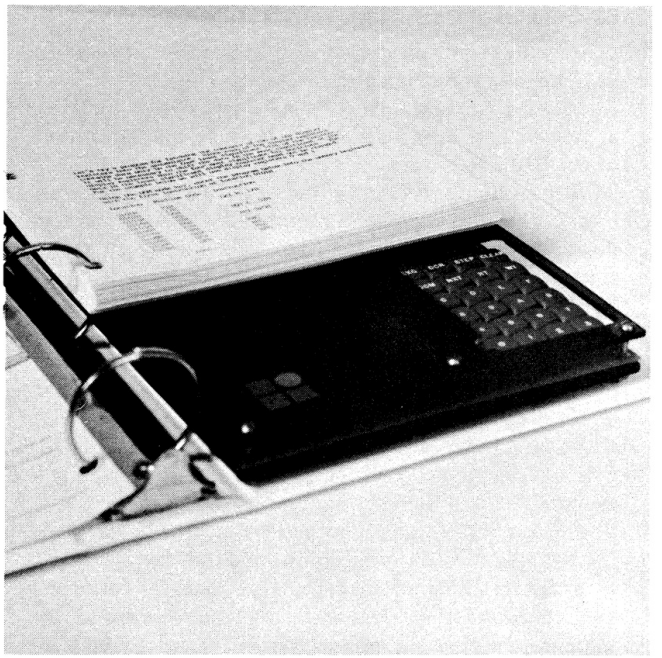
HLT – Halt

Mnemonic/Code	HLT
Operand	–
Flags Affected	None

This instruction causes the 8080 to enter an idle condition. It will remain in this state until it receives an interrupt or a rest signal.

ia7301 Computer in a Book

iasis inc.



General Information

- Programming course and operational microcomputer system enclosed in sturdy three-ring binder
- Computer system expandable to 65K bytes of memory and 256 I/O ports with optional expander boards
- Preprogrammed with permanent monitor program which operates six modes:
  1. DCM – Display and Change Memory
  2. DCR – Display and Change Register
  3. EXEC – Execute Program
  4. STEP – Single Step through Program
  5. RT – Read Tape
  6. WT – Write Tape
- Preprogrammed tapes (optional) to extend system's capabilities
- Iasis power supply (optional) PS1 provides +5V and −12V to on-board edge connector. ia7301 System generates −5V internally

Course Materials

- Over 250 pages of programming text
- Hexadecimal coding card and other programming aids

Hardware Materials

- 8080A based microcomputer system capable of executing 78 instructions
- 1024 bytes of RAM Memory
- 1024 bytes of Bipolar PROM containing the monitor program
- 24 pad keyboard with 6 modes of operation and 16 hexadecimal keys (on-board)
- Eight 7-segment LED readout displays (on-board)
- Two I/O ports and a Cassette tape recorder interface
- Three LED indicator lights showing read and write tape and execute program

General Description

Enclosed in one three-ring binder is a comprehensive programming course and fully operational 8080A based microcomputer system. The layout of the course materials and the microcomputer enables the user to read the text at the same time he operates the computer. All functions are displayed on the eight 7-segment readout displays in convenient hexadecimal code. And at all times the on-board readout displays and keyboard are readily visible and accessible.

The complete course material systematically teaches the user to write and load programs and even to edit and degug programs. The ia7301 Computer-in-a-Book introduces the fundamentals, one instruction at a time. First an application problem is introduced such as testing for an opened or closed switch. This problem is then expanded until the user has gone through several variations. As he progresses the applications become more complex, so that by the end of the course the user will be working on advanced programming problems using all the instructions and techniques he has learned.

© Iasis, Inc., 1976

The Six Modes of Operation

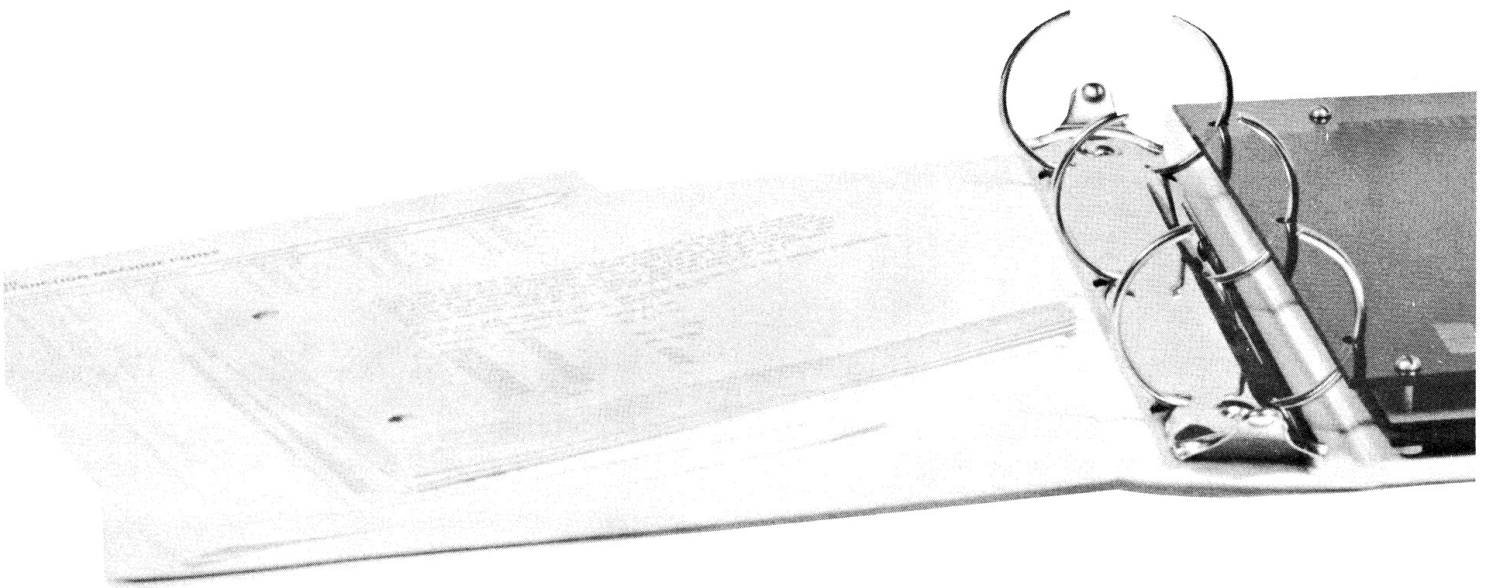
Altogether there are six modes of operation which greatly assist in loading and debugging programs. They are:

1. DCM – Display and Change Memory. The DCM mode key allows the user to retrieve, check, and change the contents of the RAM memories.
2. DCR – Display and Change Register. This mode allows the user to retrieve, check and modify the working registers, including the accumulator, flags, program counter and stack pointer.
3. STEP – Single Step Program. This mode key is one of the most useful tools in the ia7301 system. It allows the user to examine and execute each instruction of his program in operating sequence. Thus, a user can follow the execution of his program, instruction by instruction, through his entire program.
4. EXC – Execute Program. This mode allows the program to execute in a free running mode.
5. 6. RT and WT – Read Tape and Write Tape. By depressing the WT key the CPU will dump its entire contents on to a cassette tape through the tape recorder interface port. Pressing RT key will allow the tape recorder to reload the program into the microcomputer system. Two LED indicator lights monitor the dumping and reloading process so the user can see when the operation has been completed. Using the WT and RT modes, the user can build up a library of programs.

Available Preprogrammed Tapes

By obtaining preprogrammed tapes from Iasis, the user can extend his system capabilities and expand his own programming techniques. Typical tapes include a floating point arithmetic package, fascinating games of skill and some micro-assembler programs. Complete instruction text is supplied with each preprogrammed tape.

Most systems require involved start up procedures before a program can be loaded or executed. The ia7301 goes right into its monitor program whenever power is applied. This unique monitor program makes entry into the six modes of operation as easy as pressing the appropriate mode key.



Optional Expander Boards

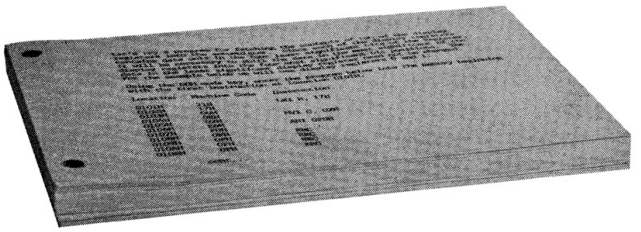
Expander boards are also available for increasing the memory of the ia7301 system and its I/O capability. Each expander board adds 4K bytes of RAM memory and six I/O ports. Thus the system can be upgraded to a full 65K bytes of memory and 256 ports.

Even though the ia7301 is packaged as an operational programming course, the computer itself can be removed from the binder and used as a complete functioning system. Its 1024 bytes of memory are large enough for the most elaborate programs and the sockets for bipolar PROMs allow the user to extend the monitor's program or store his own dedicated programs.

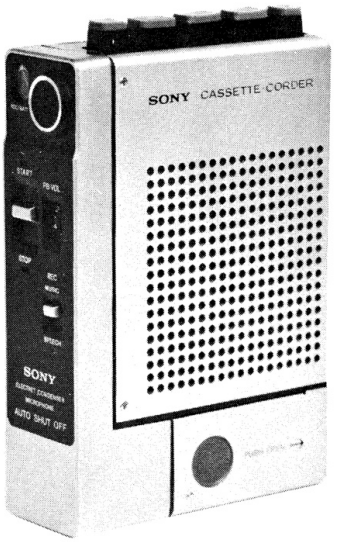
Summary

In summary the text material takes the user through the 8080A instruction set, instruction by instruction. These are combined into simple programs that become increasingly sophisticated as the course progresses. Advanced programming concepts such as indexed addressing and Jump tables are introduced. Other problems solved in the course include multi-byte multiplication routines and a set of programs for playing Pong and Mastermind on the LED readout displays. At every step, the user is keying in programs and using them to test his solutions as well as the learning programs and problems presented in the text. By the end of the course materials the user is working on advanced programs using all the instructions and techniques he has learned.

Programming text fits conveniently over keyboard and LED displays on ia7301 Computer in a Book.



Almost any consumer cassette tape will accept contents of CPU through the tape recorder interface port.



Optional iasis PS1 power supply provides +5 and –12V to on-board edge connector. ia7301 system generates –5V internally.

iasis inc.

Iasis, Inc., 815 W. Maude Ave., Sunnyvale, CA 94086 (408) 732-5700

Contents of the Design Aids

The programming aid package helps provide a quicker understanding of the concepts taught in the course, and offers a reference source for microcomputer conventions.

8008 System Sheet

A complete working system built around the 8008 Central Processing Unit (CPU) is detailed in this 17" x 22" schematic. This illustration is tied directly to the text in Volume V of the course.

8080 System Sheet

A complete system built around the 8080 CPU. Address and data busses are graphically segregated to allow maximum readability and understanding. A detailed memory layout is presented as a companion to the 8080 system. Also included are a number of pin configurations of the key components used in the 8080 system.

MCS 4/40 & 8080 Programming Pad

This 100 sheet pad will aid the student programmer in the practice of organizing and recording his own sample programs. Label, Code, Operand and Comment notations are provided for.

MCS 4/40 Architecture Sheet

This large worksheet is designed to aid you in physically monitoring the movements and operations of a program through a microcomputer system.

8080 Architecture Sheet

This sheet uses the same "Penny Box" format as the MCS 4/40 Architecture Sheet, but is configured for an 8080 system.

MCS 4/40 Pocket Reference Book

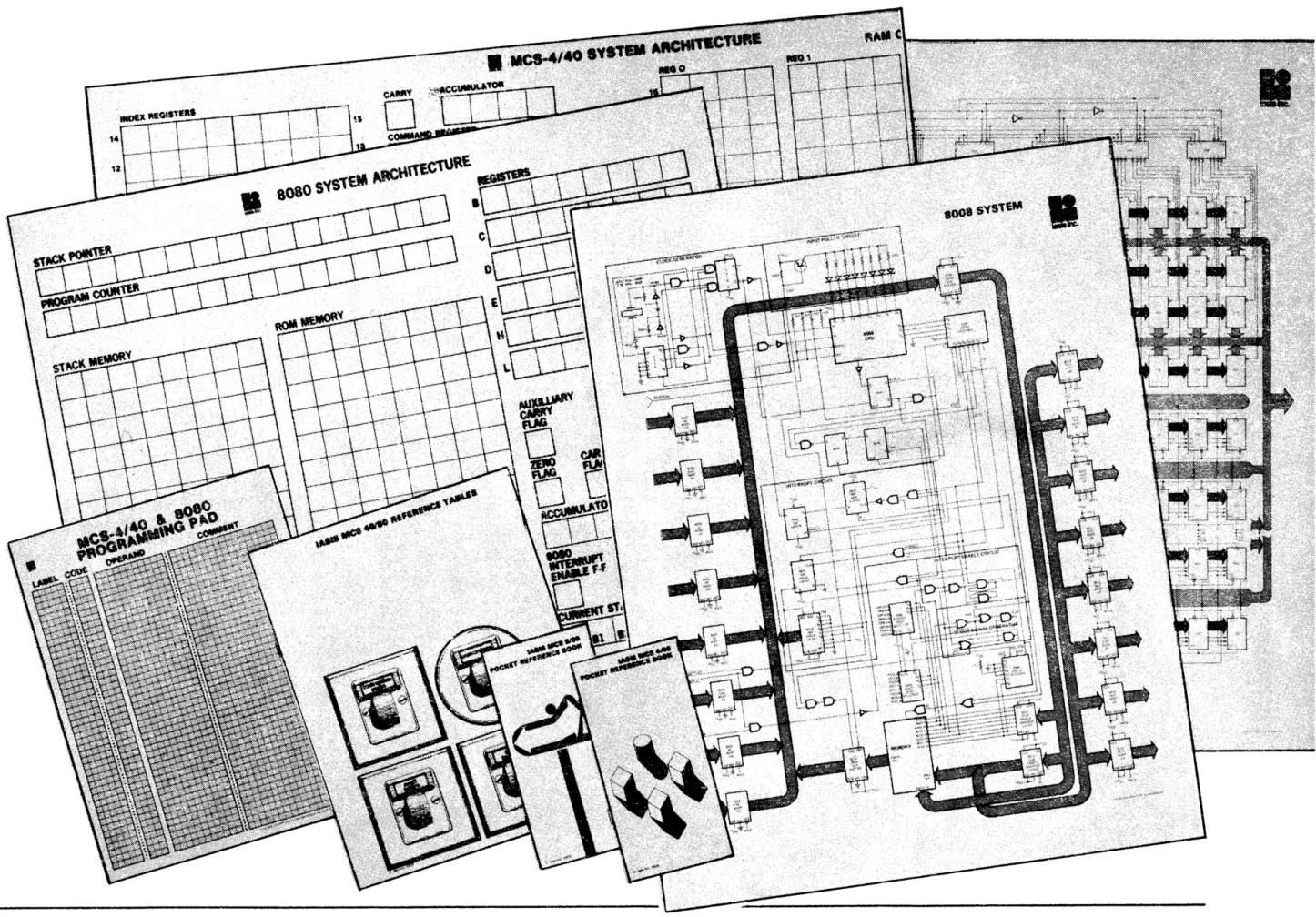
This is a pocket size reference source for the Intel 4-bit microcomputer family as discussed in Volume III of the Iasis course.

MCS 8/80 Pocket Reference Book

In the same format as the MCS 4/40 Pocket Reference Book, this aid consists of timing diagrams, pin-outs, and instruction sets and hexadecimal to mnemonic conversion tables for both the 8008 and the 8080.

Reference Tables Book

Here in one place are all of the hexadecimal to mnemonic tables for all three of the Intel microcomputers. Also included are the ASCII tables, hexadecimal addition and multiplication tables, etc.



The entire ia515 course can be purchased for $99.50 or at a discount in sets over five. Dealers and manufacturing reps. are invited to inquire about special stocking rates.

Please feel free to call or write us for further information at:

Iasis Inc., 815 West Maude Avenue, Sunnyvale, CA 94086
Phone: (408) 732-5700

IASIS SALES OFFICES

USA Office

Iasis, Inc.
815 W. Maude Avenue
Sunnyvale, California 94086
Tel: (408) 732-5700

International Offices

Belgium
ATS
Schrynbroekweg 6
3520 Zonhoven
Belgium

United Kingdom

Walmore Electronics Ltd.
11-15 Betterton Street
Drury Lane
London WC2H 9BS
Tel: 01-836-1228
TELEX: 28752

France

Tekelec Airtronic
Cite des Bruyeres
Rue Carle Vernet
92 Sevres, France
Tel: 626-02-35
TELEX: 25997

MAIL LIST CARD

If you wish to receive new Iasis literature, please fill in all the items below. (Please Print Clearly, it helps.)

Name _____ Phone _____

Company _____

Address _____

City _____ State _____ ZIP _____

MAIL TO: Iasis Inc.
815 W. Maude Ave.
Sunnyvale, California 94086

Name _____ Phone _____

Company _____

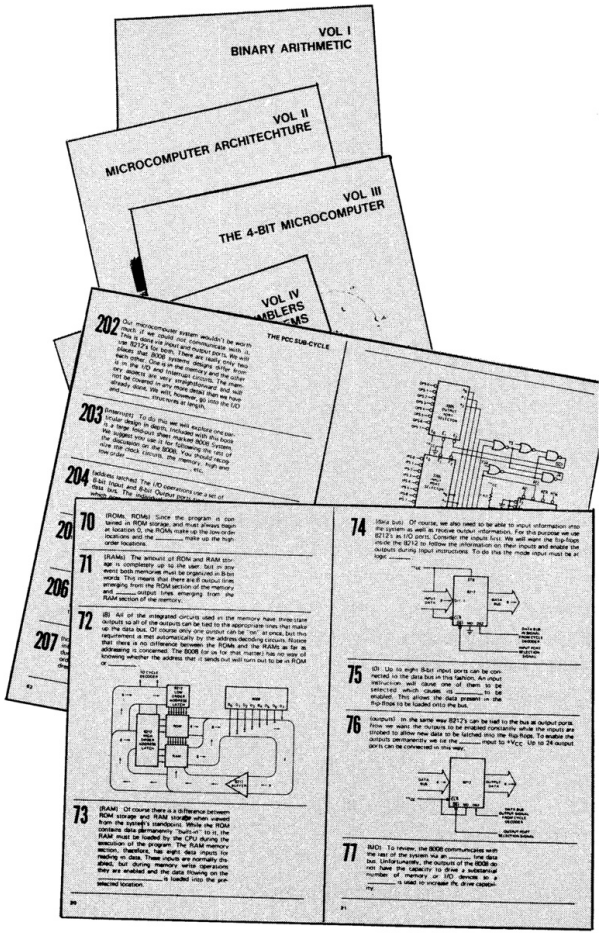
Address _____

City _____ State _____ ZIP _____

MAIL TO: Iasis Inc.
815 W. Maude Ave.
Sunnyvale, California 94086

ia515 Microcomputer Programmed Learning Course

iasis inc.



Features

- Six complete volumes—more than 700 pages of programmed learning information—covers both hardware design and programming.
- Takes you from the ABC's of microcomputers to the point where you are ready to design and program your own system.
- 1700 self tests to stay motivated and evaluate your progress.
- All essential charts, tables and design aids for a thorough understanding of microcomputers.
- Simplified and understandable format throughout.
- Step-by-step programmed instructions with all programming pads and designers references.
- The ia515 course applies to all microcomputer applications.
- Details on Intel's 4004, 4040 and 8008, 8080 microprocessors.

Description

The Iasis programmed learning series comes in six easy-to-follow volumes along with all associated design aids. This complete course combines the most effective methods of self instruction ever developed with all the technical information necessary to design a wide variety of sophisticated microcomputer systems.

The continuous learning and motivation through questions and answers is the heart of programmed learning. The Iasis course uses this stimulating technique in a way that causes the reader to devote his full attention to the entire course. It actually accomplishes this by asking the reader to participate in the learning experience by writing down the answers to carefully constructed questions. In each book of the six volume set there are 5 to 6 questions per page totaling over 1700 for the whole course. These questions are presented in a progressive sequence that provide information which will be used to answer subsequent questions. Thus the reader is continually motivated by making the learning process an active dialogue rather than simply a printed lecture. This feeling of active participation is carried throughout the course and at the conclusion, the reader will feel he has just complete a comprehensive workshop on microcomputer design and programming.

As a bonus, Iasis offers a free Microcomputer Applications Handbook to subscribers of the ia515 course. This handbook covers actual OEM system design and development and a multitude of applications for 8080 based systems.

© 1976, Iasis Inc.

Contents of the Volumes

Volume I Binary Arithmetic
72 pages, 218 frames, 14 quizzes.

The Binary Number System
Converting Binary Numbers to Decimal
Conversion of Decimal to Binary
Binary Addition
Binary Subtraction
Signed Numbers
Binary Multiplication
Binary Division
Binary Coded Decimal System
Octal Number System
The Hexadecimal Number System

Volume II Microcomputer Architecture
73 pages, 269 frames, 10 quizzes.

Memory
Registers
Program Representation in Memory
The First Instruction
The Instruction Set
Immediate Instructions
The First Program
Input/Output
Memory Reads and Writes
Jump Instructions

Volume III The 4-Bit Microcomputer
97 pages, 350 frames, 9 quizzes.

4004 CPU
The MCS-4 Instruction Set
The First Programs
Status Characters
Input/Output Ports
4308 ROM
The 4040

Volume IV Assemblers and Prototyping Systems
103 pages, 383 frames, 10 quizzes.

Programming the MCS-40
Assemblers
Index Register Instructions
Accumulator Instructions
Immediate Instructions
Transfer of Control Instructions
Memory Selection Instructions
Special 4040 Instructions
Subroutines
Interrupts
Intellec Systems

Volume V The 8-Bit Microcomputer
129 pages, 434 frames, 15 quizzes.

8008 Architecture
Timing the 8008
The System Memory
The T3 State
The PCI Sub-Cycle
The PCR Sub-Cycle
The PCW Sub-Cycle
The PCC Sub-Cycle
8008 Interrupts
The 8080 CPU
The 8080 Registers
The Wait State
The Hold State
The 8224 Clock Generator
8228 System Controller and Bus Driver
The Wait Cycle
The Interrupt System

Volume VI 8-Bit Assemblers and Compilers
116 pages, 442 frames, 9 quizzes.

Assembly Language Format
Printing Messages
The Search Program
Assembly Directives
Carry Bit Instructions
Single Register Instructions
Data Transfer Instructions
Register or Memory to Accumulator Instructions
Register Pair Instructions
Immediate Instructions
Direct Addressing Instructions
Jump Instructions
Call Subroutine Instructions
Miscellaneous Instructions
Macros

(Bonus) Microcomputer Applications Handbook
144 pages, 8 ½ x 11″

Microcomputers . . . Promises and Pitfalls
Basic Operation of the Computer
The Role of the OEM Microcomputer System
The Role of the Development System
Hardware Design Casebook
Writing the System Monitor
The ia7600 OEM System
Using the OEM System for Development
Applications
Programming the 8080

iasis, inc., 815 w. maude ave., sunnyvale, ca 94086 (408) 732-5700